Producing Open Source Software

How to Run a Successful Free Software Project

Karl Fogel

Producing Open Source Software: How to Run a Successful Free Software Project

by Karl Fogel

Dedication

This book is dedicated to two dear friends without whom it would not have been possible: Karen Underhill and Jim Blandy.

Table of Contents

Preface

Why Write This Book?

At parties, people no longer give me a blank stare when I tell them I write free software. "Oh, yes, open source—like Linux?" they say. I nod eagerly in agreement. "Yes, exactly! That's what I do." It's nice not to be completely fringe anymore. In the past, the next question was usually fairly predictable: "How do you make money doing that?" To answer, I'd summarize the economics of open source: that there are organizations in whose interest it is to have certain software exist, but that they don't need to sell copies, they just want to make sure the software is available and maintained, as a tool instead of as a commodity.

The next question is not always about money, though. The business case for open source software[1] is no longer so mysterious, and even non-programmers already understand—or at least are not surprised—that there are people employed at it full time. Instead, the next question is often "*Oh, how does that work?*"

I didn't have a satisfactory answer ready, and the harder I tried to come up with one, the more I realized how complex a topic it really is. Running a free software project is not exactly like running a business (imagine having to constantly negotiate the nature of your product with a group of random people of diverse motivations and interests, most of whom you've never met!). Nor, for various reasons, is it exactly like running a traditional non-profit organization, nor a government. It has similarities to all these things, but I have slowly come to the conclusion that free software is *sui generis*. There are many things with which it can be usefully compared, but none with which it can be equated. Indeed, even the assumption that free software projects can be "run" is a stretch. A free software project can be *started*, and it can be influenced by interested parties, often quite strongly. But its assets cannot be made the property of any single owner, and as long as there are people somewhere—anywhere—interested in continuing it, it cannot be unilaterally shut down. Everyone has infinite power; everyone has no power. It's an interesting dynamic.

That is why I wanted to write this book in the first place, and seven years later, wanted to update it. Free software projects have evolved a distinct culture, an ethos in which the liberty to make the software do anything one wants is a central tenet. Yet the result of this liberty is not a scattering of individuals each going their own separate way with the code, but enthusiastic collaboration. Indeed, competence at cooperation itself is one of the most highly valued skills in free software. To manage these projects is to engage in a kind of hypertrophied cooperation, where one's ability not only to work with others but to come up with new ways of working together can result in tangible benefits to the software. This book attempts to describe the techniques by which this may be done. It is by no means complete, but it is at least a beginning.

Good free software is a worthy goal in itself, and I hope that readers who come looking for ways to achieve it will be satisfied with what they find here. But beyond that I also hope to convey something of the sheer pleasure to be had from working with a motivated team of open source developers, and from interacting with users in the wonderfully direct way that open source encourages. Participating in a successful free software project is a deep pleasure, and ultimately that's what keeps the whole system going.

Who Should Read This Book?

This book is meant for software developers and managers who are considering starting an open source project, or who have started one and are wondering what to do now. It should also be helpful for people who just want to participate in an open source project but have never done so before.

[1] The terms "open source software" and "free software" are essentially synonymous in this context; they are discussed more in the section called ""Free" Versus "Open Source"" in Chapter 1, *Introduction*.

The reader need not be a programmer, but should know basic software engineering concepts such as APIs, source code, compilers, and patches.

Prior experience with open source software, as either a user or a developer, is not necessary. Those who have worked in free software projects before will probably find at least some parts of the book a bit obvious, and may want to skip those sections. Because there's such a potentially wide range of audience experience, I've made an effort to label sections clearly, and to say when something can be skipped by those already familiar with the material.

Sources

Much of the raw material for the first edition of this book came from five years of working with the Subversion project (subversion.tigris.org [http://subversion.apache.org/]). Subversion is an open source version control system, written from scratch, and intended to replace CVS as the *de facto* version control system of choice in the open source community. The project was started by my employer, CollabNet (collab.net [http://www.collab.net/]), in early 2000, and thank goodness CollabNet understood right from the start how to run it as a truly collaborative, distributed effort. We got a lot of developer buy-in early on; today there are 50-some developers on the project, of whom only a few are CollabNet employees.

Subversion is in many ways a classic example of an open source project, and I ended up drawing on it more heavily than I originally expected. This was partly a matter of convenience: whenever I needed an example of a particular phenomenon, I could usually call one up from Subversion right off the top of my head. But it was also a matter of verification. Although I am involved in other free software projects to varying degrees, and talk to friends and acquaintances involved in many more, one quickly realizes when writing for print that all assertions need to be fact-checked. I didn't want to make statements about events in other projects based only on what I could read in their public mailing list archives. If someone were to try that with Subversion, I knew, she'd be right about half the time and wrong the other half. So when drawing inspiration or examples from a project with which I didn't have direct experience, I tried to first talk to an informant there, someone I could trust to explain what was really going on.

While Subversion was my full time job from 2000-2006, I've been involved in free software for more than twenty years, including the seven years since 2006 (when the first edition of this book was published). Other projects and organizations that have influenced this book include:

- The GNU Emacs text editor project at the Free Software Foundation, in which I maintain a few small packages.
- Concurrent Versions System (CVS), which I worked on intensely in 1994–1995 with Jim Blandy and was involved with intermittently for a few years afterwards.
- The collection of open source projects known as the Apache Software Foundation, especially the Apache Portable Runtime (APR) and Apache HTTP Server.
- The Launchpad.net project at Canonical, Ltd.
- Code for America and O'Reilly Media, which gave me an inside view on open source civic technology development starting in 2010, and kindly kept me in the loop after I became a full-time independent consultant in 2012.
- The many open source anti-surveillance and censorship-circumvention tools supported by the Open Internet Tools Project (OpenITP.org) and by the Open Technology Institute at the New America Foundation.
- Checkbook NYC, the municipal financial transparency software released by the New York City Office of the Comptroller.

- The Arches Project, an open source geospatial web application for inventorying and helping protect cultural heritage sites (e.g., historic buildings, archeological sites, etc), created by the Getty Conservation Institute and World Monuments Fund.
- OpenOffice.org / LibreOffice.org, the Berkeley Database from Sleepycat, and MySQL Database; I have not been involved with these projects personally, but have observed them and, in some cases, talked to people there.
- GNU Debugger (GDB) (likewise).
- The Debian Project (likewise).
- The Hypothes.is Project (likewise).

This is not a complete list, of course. Many of the client projects I work with through our consulting practice at at Open Tech Strategies, LLC [http://opentechstrategies.com/] have influenced this book, and like most open source programmers, I also keep loose tabs on a variety of different projects of interest to me, just to have a sense of the general state of things. I haven't named all of them here, but they are mentioned in the text where appropriate.

Acknowledgments

This book took four times longer to write than I thought it would, and for much of that time felt rather like a grand piano suspended above my head wherever I went. Without help from many people, I would not have been able to complete it while staying sane.

Andy Oram, my editor at O'Reilly, was a writer's dream. Aside from knowing the field intimately (he suggested many of the topics), he has the rare gift of knowing what one meant to say and helping one find the right way to say it. It has been an honor to work with him. Thanks also to Chuck Toporek for steering this proposal to Andy right away.

Brian Fitzpatrick reviewed almost all of the material as I wrote it, which not only made the book better, but kept me writing when I wanted to be anywhere in the world but in front of the computer. Ben Collins-Sussman and Mike Pilato also checked up on progress, and were always happy to discuss—sometimes at length—whatever topic I was trying to cover that week. They also noticed when I slowed down, and gently nagged when necessary. Thanks, guys.

Biella Coleman was writing her dissertation at the same time I was writing this book. She knows what it means to sit down and write every day, and provided an inspiring example as well as a sympathetic ear. She also has a fascinating anthropologist's-eye view of the free software movement, giving both ideas and references that I was able use in the book. Alex Golub—another anthropologist with one foot in the free software world, and also finishing his dissertation at the same time—was exceptionally supportive early on, which helped a great deal.

Micah Anderson somehow never seemed too oppressed by his own writing gig, which was inspiring in a sick, envy-generating sort of way, but he was ever ready with friendship, conversation, and (on at least one occasion) technical support. Thanks, Micah!

Jon Trowbridge and Sander Striker gave both encouragement and concrete help—their broad experience in free software provided material I couldn't have gotten any other way.

Thanks to Greg Stein not only for friendship and well-timed encouragement, but for showing the Subversion project how important regular code review is in building a programming community. Thanks also to Brian Behlendorf, who tactfully drummed into our heads the importance of having discussions publicly; I hope that principle is reflected throughout this book.

Thanks to Benjamin "Mako" Hill and Seth Schoen, for various conversations about free software and its politics; to Zack Urlocker and Louis Suarez-Potts for taking time out of their busy schedules to be interviewed; to Shane on the Slashcode list for allowing his post to be quoted; and to Haggen So for his enormously helpful comparison of canned hosting sites.

Thanks to Alla Dekhtyar, Polina, and Sonya for their unflagging and patient encouragement. I'm very glad that I will no longer have to end (or rather, try unsuccessfully to end) our evenings early to go home and work on "The Book."

Thanks to Jack Repenning for friendship, conversation, and a stubborn refusal to ever accept an easy wrong analysis when a harder right one is available. I hope that some of his long experience with both software development and the software industry rubbed off on this book.

CollabNet was exceptionally generous in allowing me a flexible schedule to write, and didn't complain when it went on far longer than originally planned. I don't know all the intricacies of how management arrives at such decisions, but I suspect Sandhya Klute, and later Mahesh Murthy, had something to do with it—my thanks to them both.

The entire Subversion development team has been an inspiration for the past five years, and much of what is in this book I learned from working with them. I won't thank them all by name here, because there are too many, but I implore any reader who runs into a Subversion committer to immediately buy that committer the drink of his choice—I certainly plan to.

Many times I ranted to Rachel Scollon about the state of the book; she was always willing to listen, and somehow managed to make the problems seem smaller than before we talked. That helped a lot—thanks.

Thanks (again) to Noel Taylor, who must surely have wondered why I wanted to write another book given how much I complained the last time, but whose friendship and leadership of Golosá helped keep music and good fellowship in my life even in the busiest times. Thanks also to Matthew Dean and Dorothea Samtleben, friends and long-suffering musical partners, who were very understanding as my excuses for not practicing piled up. Megan Jennings was constantly supportive, and genuinely interested in the topic even though it was unfamiliar to her—a great tonic for an insecure writer. Thanks, pal!

I had four knowledgeable and diligent reviewers for this book: Yoav Shapira, Andrew Stellman, Davanum Srinivas, and Ben Hyde. If I had been able to incorporate all of their excellent suggestions, this would be a better book. As it was, time constraints forced me to pick and choose, but the improvements were still significant. Any errors that remain are entirely my own.

My parents, Frances and Henry, were wonderfully supportive as always, and as this book is less technical than the previous one, I hope they'll find it somewhat more readable.

Finally, I would like to thank the dedicatees, Karen Underhill and Jim Blandy. Karen's friendship and understanding have meant everything to me, not only during the writing of this book but for the last seven years. I simply would not have finished without her help. Likewise for Jim, a true friend and a hacker's hacker, who first taught me about free software, much as a bird might teach an airplane about flying.

Disclaimer

The thoughts and opinions expressed in this book are my own. They do not necessarily represent the views of my clients, past employers, the New America Foundation, or the open source projects discussed here. They do, however, represent the views of Jim Blandy. Seriously: he agrees with everything in this book. Ask him.

Chapter 1. Introduction

Most free software projects fail.

We tend not to hear very much about the failures. Only successful projects attract attention, and there are so many free software projects in total[1] that even though only a small percentage succeed, the result is still a lot of visible projects. We also don't hear about the failures because failure is not an event. There is no single moment when a project ceases to be viable; people just sort of drift away and stop working on it. There may be a moment when a final change is made to the project, but those who made it usually didn't know at the time that it was the last one. There is not even a clear definition of when a project is expired. Is it when it hasn't been actively worked on for six months? When its user base stops growing, without having exceeded the developer base? What if the developers of one project abandon it because they realized they were duplicating the work of another—and what if they join that other project, then expand it to include much of their earlier effort? Did the first project end, or just change homes?

Because of such complexities, it's impossible to put a precise number on the failure rate. But anecdotal evidence from two decades in open source, some casting around on multi-project hosting sites (see the section called “Canned Hosting” in Chapter 3, *Technical Infrastructure* for more about them), and a little Googling all point to the same conclusion: the rate is extremely high, probably on the order of 90–95%. The number climbs higher if you include surviving but dysfunctional projects: those which *are* producing running code, but which are not pleasant places to be, or are not making progress as quickly or as dependably as they could.

This book is about avoiding failure. It examines not only how to do things right, but how to do them wrong, so you can recognize and correct problems early. My hope is that after reading it, you will have a repertory of techniques not just for avoiding common pitfalls of open source development, but also for dealing with the growth and maintenance of a successful project. Success is not a zero-sum game, and this book is not about winning or getting ahead of the competition. Indeed, an important part of running an open source project is working smoothly with other, related projects. In the long run, every successful project contributes to the well-being of the overall, worldwide body of free software.

It would be tempting to say that free software projects fail for the same sorts of reasons proprietary software projects do. Certainly, free software has no monopoly on unrealistic requirements, vague specifications, poor resource management, insufficient design phases, or any of the other hobgoblins already well known to the software industry. There is a huge body of writing on these topics, and I will try not to duplicate it in this book. Instead, I will attempt to describe the problems peculiar to free software. When a free software project runs aground, it is often because the developers (or the managers) did not appreciate the unique problems of open source software development, even though they might be quite well-prepared for the better-known difficulties of closed-source development.

One of the most common mistakes is unrealistic expectations about the benefits of open source itself. An open license does not guarantee that hordes of active developers will suddenly devote their time to your project, nor does open-sourcing a troubled project automatically cure its ills. In fact, quite the opposite: opening up a project can add whole new sets of complexities, and cost *more* in the short term than simply keeping it in-house. Opening up means arranging the code to be comprehensible to complete strangers, setting up development documentation and email lists, and often writing documentation for the first time. All this is a lot of work. And of course, if any interested developers *do* show up, there is the added burden of answering their questions for a while before seeing any benefit from their presence. As developer Jamie Zawinski said about the troubled early days of the Mozilla project:

[1] I tried to estimate the number, by looking at just the number of projects registered at the most popular hosting sites, and the closest I could calculate to an answer was somewhere between one hundred thousand and two hundred thousand. That would still be far lower the total number of free software projects on the Internet, of course, as it only counts the ones that chose to use one of the major hosting sites.

> *Open source does work, but it is most definitely not a panacea. If there's a cautionary tale here, it is that you can't take a dying project, sprinkle it with the magic pixie dust of "open source," and have everything magically work out. Software is hard. The issues aren't that simple.*
>
> (from **jwz.org/gruntle/nomo.html [https://www.jwz.org/gruntle/nomo.html]**)

A related mistake is that of skimping on presentation and packaging, figuring that these can always be done later, when the project is well under way. Presentation and packaging comprise a wide range of tasks, all revolving around the theme of reducing the barrier to entry. Making the project inviting to the uninitiated means writing user and developer documentation, setting up docs that are informative to newcomers, automating as much of the software's compilation and installation as possible, etc.

Many programmers unfortunately treat this kind of work as being of secondary importance to the code itself. There are a couple of reasons for this. First, it can feel like busywork, because its benefits are most visible to those least familiar with the project — and vice versa: after all, the people who develop the code don't really need the packaging. They already know how to install, administer, and use the software, because they wrote it. Second, the skills required to do presentation and packaging well are often completely different from those required to write code. People tend to focus on what they're good at, even if it might serve the project better to spend a little time on something that suits them less. Chapter 2, *Getting Started* discusses presentation and packaging in detail, and explains why it's crucial that they be a priority from the very start of the project.

Next comes the fallacy that little or no project management is required in open source, or conversely, that the same management practices used for in-house development will work equally well on an open source project. Management in an open source project isn't always very visible, but in the successful projects, it's usually happening behind the scenes in some form or another. A small thought experiment suffices to show why. An open source project consists of a random collection of programmers—already a notoriously independent-minded species—who have most likely never met each other, and who may each have different personal goals in working on the project. The thought experiment is simply to imagine what would happen to such a group *without* management. Barring miracles, it would collapse or drift apart very quickly. Things won't simply run themselves, much as we might wish otherwise. But the management, though it may be quite active, is often informal, subtle, and low-key. The only thing keeping a development group together is their shared belief that they can do more in concert than individually. Thus the goal of management is mostly to ensure that they continue to believe this, by setting standards for communications, by making sure useful developers don't get marginalized due to personal idiosyncracies, and in general by making the project a place developers want to keep coming back to. Specific techniques for doing this are discussed throughout the rest of this book.

Finally, there is a general category of problems that may be called "failures of cultural navigation." Twenty years ago, even ten, it would have been premature to talk about a global culture of free software, but not anymore. A recognizable culture has slowly emerged, and while it is certainly not monolithic—it is at least as prone to internal dissent and factionalism as any geographically bound culture—it does have a basically consistent core. Most successful open source projects exhibit some or all of the characteristics of this core. They reward certain types of behaviors, and punish others; they create an atmosphere that encourages unplanned participation, sometimes at the expense of central coordination; they have concepts of rudeness and politeness that can differ substantially from those prevalent elsewhere. Most importantly, longtime participants have generally internalized these standards, so that they share a rough consensus about expected conduct. Unsuccessful projects usually deviate in significant ways from this core, albeit unintentionally, and often do not have a consensus about what constitutes reasonable default behavior. This means that when problems arise, the situation can quickly deteriorate, as the participants lack an already established stock of cultural reflexes to fall back on for resolving differences.

That last category, failures of cultural navigation, includes an interesting phenomenon: certain types of organizations are structurally less compatible with open source development than others. One of the

great surprises for me in preparing the second edition of this book was realizing that, on the whole, my experiences indicated that governments are less naturally suited to participating in free software projects than private-sector, for-profit corporations, with non-profits somewhere in between the two. There are many reasons for this (see the section called “Governments and Open Source”), and the problems are certainly surmountable, but it's worth noting that when an existing organization — particularly a hierarchical one, and *particularly* a hierarchical, risk-averse, and publicity-sensitive one — starts or joins an open source project, some adjustments will usually be needed.

This book is a practical guide, not an anthropological study or a history. However, a working knowledge of the origins of today's free software culture is an essential foundation for any practical advice. A person who understands the culture can travel far and wide in the open source world, encountering many local variations in custom and dialect, yet still be able to participate comfortably and effectively everywhere. In contrast, a person who does not understand the culture will find the process of organizing or participating in a project difficult and full of surprises. Since the number of people developing free software is still growing by leaps and bounds, there are many people in that latter category—this is largely a culture of recent immigrants, and will continue to be so for some time. If you think you might be one of them, the next section provides background for discussions you'll encounter later, both in this book and on the Internet. (On the other hand, if you've been working with open source for a while, you may already know a lot of its history, so feel free to skip the next section.)

History

Software sharing has been around as long as software itself. In the early days of computers, manufacturers felt that competitive advantages were to be had mainly in hardware innovation, and therefore didn't pay much attention to software as a business asset. Many of the customers for these early machines were scientists or technicians, who were able to modify and extend the software shipped with the machine themselves. Customers sometimes distributed their patches back not only to the manufacturer, but to other owners of similar machines. The manufacturers often tolerated and even encouraged this: in their eyes, improvements to the software, from whatever source, just made the hardware more attractive to other potential customers.

Although this early period resembled today's free software culture in many ways, it differed in two crucial respects. First, there was as yet little standardization of hardware—it was a time of flourishing innovation in computer design, but the diversity of computing architectures meant that everything was incompatible with everything else. Software written for one machine would generally not work on another; programmers tended to acquire expertise in a particular architecture or family of architectures (whereas today they would be more likely to acquire expertise in a programming language or family of languages, confident that their expertise will be transferable to whatever computing hardware they happen to find themselves working with). Because a person's expertise tended to be specific to one kind of computer, their accumulation of expertise had the effect of making that particular architecture computer more attractive to them and their colleagues. It was therefore in the manufacturer's interests for machine-specific code and knowledge to spread as widely as possible.

Second, there was no widespread Internet. Though there were fewer legal restrictions on sharing than there are today, the technical restrictions were greater: the means of getting data from place to place were inconvenient and cumbersome, relatively speaking. There were some small, local networks, good for sharing information among employees at the same lab or company. But there remained barriers to overcome if one wanted to share with the world. These barriers *were* overcome in many cases. Sometimes different groups made contact with each other independently, sending disks or tapes through land mail, and sometimes the manufacturers themselves served as central clearing houses for patches. It also helped that many of the early computer developers worked at universities, where publishing one's knowledge was expected. But the physical realities of data transmission meant there was always an impedance to sharing, an impedance proportional to the distance (real or organizational) that the software had to travel. Widespread, frictionless sharing, as we know it today, was not possible.

The Rise of Proprietary Software and Free Software

As the industry matured, several interrelated changes occurred simultaneously. The wild diversity of hardware designs gradually gave way to a few clear winners—winners through superior technology, superior marketing, or some combination of the two. At the same time, and not entirely coincidentally, the development of so-called "high level" programming languages meant that one could write a program once, in one language, and have it automatically translated ("compiled") to run on different kinds of computers. The implications of this were not lost on the hardware manufacturers: a customer could now undertake a major software engineering effort without necessarily locking themselves into one particular computer architecture. When this was combined with the gradual narrowing of performance differences between various computers, as the less efficient designs were weeded out, a manufacturer that treated its hardware as its only asset could look forward to a future of declining profit margins. Raw computing power was becoming a fungible good, while software was becoming the differentiator. Selling software, or at least treating it as an integral part of hardware sales, began to look like a good strategy.

This meant that manufacturers had to start enforcing the copyrights on their code more strictly. If users simply continued to share and modify code freely among themselves, they might independently reimplement some of the improvements now being sold as "added value" by the supplier. Worse, shared code could get into the hands of competitors. The irony is that all this was happening around the time the Internet was getting off the ground. So just when truly unobstructed software sharing was finally becoming technically possible, changes in the computer business made it economically undesirable, at least from the point of view of any single company. The suppliers clamped down, either denying users access to the code that ran their machines, or insisting on non-disclosure agreements that made effective sharing impossible.

Conscious resistance

As the world of unrestricted code swapping slowly faded away, a counterreaction crystallized in the mind of at least one programmer. Richard Stallman worked in the Artificial Intelligence Lab at the Massachusetts Institute of Technology in the 1970s and early '80s, during what turned out to be a golden age and a golden location for code sharing. The AI Lab had a strong "hacker ethic",[2] and people were not only encouraged but expected to share whatever improvements they made to the system. As Stallman wrote later:

> *We did not call our software "free software", because that term did not yet exist; but that is what it was. Whenever people from another university or a company wanted to port and use a program, we gladly let them. If you saw someone using an unfamiliar and interesting program, you could always ask to see the source code, so that you could read it, change it, or cannibalize parts of it to make a new program.*
>
> (from **gnu.org/gnu/thegnuproject.html [https://www.gnu.org/gnu/thegnuproject.html]**)

This Edenic community collapsed around Stallman shortly after 1980, when the changes that had been happening in the rest of the industry finally caught up with the AI Lab. A startup company hired away many of the Lab's programmers to work on an operating system similar to what they had been working on at the Lab, only now under an exclusive license. At the same time, the AI Lab acquired new equipment that came with a proprietary operating system.

Stallman saw the larger pattern in what was happening:

[2] Stallman uses the word "hacker" in the sense of "someone who loves to program and enjoys being clever about it," not the somewhat newer meaning of "someone who breaks into computers."

> *The modern computers of the era, such as the VAX or the 68020, had their own operating systems, but none of them were free software: you had to sign a nondisclosure agreement even to get an executable copy.*
>
> *This meant that the first step in using a computer was to promise not to help your neighbor. A cooperating community was forbidden. The rule made by the owners of proprietary software was, "If you share with your neighbor, you are a pirate. If you want any changes, beg us to make them."*

By some quirk of personality, he decided to resist the trend. Instead of continuing to work at the now-decimated AI Lab, or taking a job writing code at one of the new companies, where the results of his work would be kept locked in a box, he resigned from the Lab and started the GNU Project and the Free Software Foundation (FSF). The goal of GNU[3] was to develop a completely free and open computer operating system and body of application software, in which users would never be prevented from hacking or from sharing their modifications. He was, in essence, setting out to recreate what had been destroyed at the AI Lab, but on a world-wide scale and without the vulnerabilities that had made the AI Lab's culture susceptible to disintegration.

In addition to working on the new operating system, Stallman devised a copyright license whose terms guaranteed that his code would be perpetually free. The GNU General Public License (GPL) is a clever piece of legal judo: it says that the code may be copied and modified without restriction, and that both copies and derivative works (i.e., modified versions) must be distributed under the same license as the original, with no additional restrictions. In effect, it uses copyright law to achieve an effect opposite to that of traditional copyright: instead of limiting the software's distribution, it prevents *anyone*, even the author, from limiting distribution. For Stallman, this was better than simply putting his code into the public domain. If it were in the public domain, any particular copy of it could be incorporated into a proprietary program (as also sometimes happens to code under permissive open source copyright licenses [4]). While such incorporation wouldn't in any way diminish the original code's continued availability, it would have meant that Stallman's efforts could benefit the enemy—proprietary software. The GPL can be thought of as a form of protectionism for free software, because it prevents non-free software from taking full advantage of GPLed code. The GPL and its relationship to other free software licenses are discussed in detail in Chapter 9, *Legal Matters: Licenses, Copyrights, Trademarks and Patents*.

With the help of many programmers, some of whom shared Stallman's ideology and some of whom simply wanted to see a lot of free code available, the GNU Project began releasing free replacements for many of the most critical components of an operating system. Because of the now-widespread standardization in computer hardware and software, it was possible to use the GNU replacements on otherwise non-free systems, and many people did. The GNU text editor (Emacs) and C compiler (GCC) were particularly successful, gaining large and loyal followings not on ideological grounds, but simply on their technical merits. By about 1990, GNU had produced most of a free operating system, except for the kernel—the part that the machine actually boots up, and that is responsible for managing memory, disk, and other system resources.

Unfortunately, the GNU project had chosen a kernel design that turned out to be harder to implement than expected. The ensuing delay prevented the Free Software Foundation from making the first release of an entirely free operating system. The final piece was put into place instead by Linus Torvalds, a Finnish computer science student who, with the help of developers around the world, had completed a free kernel using a more conservative design. He named it Linux, and when it was combined with the existing GNU programs and other free software (especially the X Windows System), the result was

[3] It stands for "GNU's Not Unix", and the "GNU" in that expansion stands for an infinitely long footnote.
[4] See the section called “Terminology” for more about "permissive" licensing versus GPL-style "copyleft" licensing. The opensource.org FAQ is also a good resource on this—seeopensource.org/faq#copyleft [https://opensource.org/faq#copyleft].

a completely free operating system. For the first time, you could boot up your computer and do work without using any proprietary software.[5]

Much of the software on this new operating system was not produced by the GNU project. In fact, GNU wasn't even the only group working on producing a free operating system (for example, the code that eventually became NetBSD and FreeBSD was already under development by this time). The importance of the Free Software Foundation was not only in the code they wrote, but in their political rhetoric. By talking about free software as a cause instead of a convenience, they made it difficult for programmers *not* to have a political consciousness about it. Even those who disagreed with the FSF had to engage the issue, if only to stake out a different position. The FSF's effectiveness as propagandists lay in tying their code to a message, by means of the GPL and other texts. As their code spread widely, that message spread as well.

Accidental resistance

There were many other things going on in the nascent free software scene, however, and not all were as explictly ideological as Stallman's GNU Project. One of the most important was the *Berkeley Software Distribution* (*BSD*), a gradual re-implementation of the Unix operating system—which up until the late 1970's had been a loosely proprietary research project at AT&T—by programmers at the University of California at Berkeley. The BSD group did not make any overt political statements about the need for programmers to band together and share with one another, but they *practiced* the idea with flair and enthusiasm, by coordinating a massive distributed development effort in which the Unix command-line utilities and code libraries, and eventually the operating system kernel itself, were rewritten from scratch mostly by volunteers. The BSD project became a prime example of non-ideological free software development, and also served as a training ground for many developers who would go on to remain active in the open source world.

Another crucible of cooperative development was the *X Window System*, a free, network-transparent graphical computing environment, developed at MIT in the mid-1980's in partnership with hardware vendors who had a common interest in being able to offer their customers a windowing system. Far from opposing proprietary software, the X license deliberately allowed proprietary extensions on top of the free core—each member of the consortium wanted the chance to enhance the default X distribution, and thereby gain a competitive advantage over the other members. X Windows[6] itself was free software, but mainly as a way to level the playing field between competing business interests and increase standardization, not out of some desire to end the dominance of proprietary software. Yet another example, predating the GNU project by a few years, was TeX, Donald Knuth's free, publishing-quality typesetting system. He released it under terms that allowed anyone to modify and distribute the code, but not to call the result "TeX" unless it passed a very strict set of compatibility tests (this is an example of the "trademark-protecting" class of free licenses, discussed more in Chapter 9, *Legal Matters: Licenses, Copyrights, Trademarks and Patents*). Knuth wasn't taking a stand one way or the other on the question of free-versus-proprietary software; he just needed a better typesetting system in order to complete his *real* goal—a book on computer programming—and saw no reason not to release his system to the world when done.

Without listing every project and every license, it's safe to say that by the late 1980's, there was a lot of free software available under a wide variety of licenses. The diversity of licenses reflected a corresponding diversity of motivations. Even some of the programmers who chose the GNU GPL were much less ideologically driven than the GNU project itself was. Although they enjoyed working on free software, many developers did not consider proprietary software a social evil. There were people who felt a moral impulse to rid the world of "software hoarding" (Stallman's term for non-free software), but others were

[5]Technically, Linux was not the first. A free operating system for IBM-compatible computers, called 386BSD, had come out shortly before Linux. However, it was a lot harder to get 386BSD up and running. Linux made such a splash not only because it was free, but because it actually had a high chance of successfully booting your computer after you installed it.

[6]They prefer it to be called the "X Window System", but in practice, people usually call it "X Windows", because three words is just too cumbersome.

motivated more by technical excitement, or by the pleasure of working with like-minded collaborators, or even by a simple human desire for glory. Yet by and large these disparate motivations did not interact in destructive ways. This may be because software, unlike other creative forms like prose or the visual arts, must pass semi-objective tests in order to be considered successful: it must run, and be reasonably free of bugs. This gives all participants in a project a kind of automatic common ground, a reason and a framework for working together without worrying too much about qualifications or motivations beyond the technical.

Developers had another reason to stick together as well: it turned out that the free software world was producing some very high-quality code. In some cases, it was demonstrably technically superior to the nearest non-free alternative; in others, it was at least comparable, and of course it always cost less. While only a few people might have been motivated to run free software on strictly philosophical grounds, a great many people were happy to run it because it did a better job. And of those who used it, some percentage were always willing to donate their time and skills to help maintain and improve the software.

This tendency to produce good code was certainly not universal, but it was happening with increasing frequency in free software projects around the world. Businesses that depended heavily on software gradually began to take notice. Many of them discovered that they were already using free software in day-to-day operations, and simply hadn't known it (upper management isn't always aware of everything the IT department does). Corporations began to take a more active and public role in free software projects, contributing time and equipment, and sometimes even directly funding the development of free programs. Such investments could, in the best scenarios, repay themselves many times over. The sponsor only pays a small number of expert programmers to devote themselves to the project full time, but reaps the benefits of *everyone's* contributions, including work from programmers being paid by other corporations and from volunteers who have their own disparate motivations.

"Free" Versus "Open Source"

As the corporate world gave more and more attention to free software, programmers were faced with new issues of presentation. One was the word "free" itself. On first hearing the term "free software" many people mistakenly think it means just "zero-cost software." It's true that all free software is zero-cost,[7] but not all zero-cost software is free as in "freedom"—that is, the freedom to share and modify for any purpose. For example, during the battle of the browsers in the 1990s, both Netscape and Microsoft gave away their competing web browsers at no charge, in a scramble to gain market share. Neither browser was free in the "free software" sense. You couldn't get the source code, and even if you could, you didn't have the right to modify or redistribute it.[8] The only thing you could do was download an executable and run it. The browsers were no more free than shrink-wrapped software bought in a store; they merely had a lower price.

This confusion over the word "free" is due entirely to an unfortunate ambiguity in the English language. Most other tongues distinguish low prices from liberty (the distinction between *gratis* and *libre* is immediately clear to speakers of Romance languages, for example). But English's position as the de facto bridge language of the Internet means that a problem with English is, to some degree, a problem for everyone. The misunderstanding around the word "free" was so prevalent that free software programmers eventually evolved a standard formula in response: "It's *free* as in *freedom*—think *free speech*, not *free beer*." Still, having to explain it over and over is tiring. Many programmers felt, with some justification, that the ambiguous word "free" was hampering the public's understanding of this software.

But the problem went deeper than that. The word "free" carried with it an inescapable moral connotation: if freedom was an end in itself, it didn't matter whether free software also happened to be better, or

[7] One may charge a fee for giving out copies of free software, but since one cannot stop the recipients from offering it at no charge afterwards, the price is effectively driven to zero immediately.

[8] The source code to Netscape Navigator *was* eventually released under an open source license, in 1998, and became the foundation for the Mozilla web browser. See mozilla.org [https://www.mozilla.org/].

more profitable for certain businesses in certain circumstances. Those were merely pleasant side effects of a motive that was, at its root, neither technical nor mercantile, but moral. Furthermore, the "free as in freedom" position forced a glaring inconsistency on corporations who wanted to support particular free programs in one aspect of their business, but continue marketing proprietary software in others.

These dilemmas came to a community that was already poised for an identity crisis. The programmers who actually *write* free software have never been of one mind about the overall goal, if any, of the free software movement. Even to say that opinions run from one extreme to the other would be misleading, in that it would falsely imply a linear range where there is instead a multidimensional scattering. However, two broad categories of belief can be distinguished, if we are willing to ignore subtleties for the moment. One group takes Stallman's view, that the freedom to share and modify is the most important thing, and that therefore if you stop talking about freedom, you've left out the core issue. Others feel that the software itself is the most important argument in its favor, and are uncomfortable with proclaiming proprietary software inherently bad. Some, but not all, free software programmers believe that the author (or employer, in the case of paid work) *should* have the right to control the terms of distribution, and that no moral judgement need be attached to the choice of particular terms. Others don't believe this.

For a long time, these differences did not need to be carefully examined or articulated, but free software's burgeoning success in the business world made the issue unavoidable. In 1998, the term *open-source* was created as an alternative to "free", by a coalition of programmers who eventually became The Open Source Initiative (OSI).[9] The OSI felt not only that "free software" was potentially confusing, but that the word "free" was just one symptom of a general problem: that the movement needed a marketing program to pitch it to the corporate world, and that talk of morals and the social benefits of sharing would never fly in corporate boardrooms. In their own words at the time:

> *The Open Source Initiative is a marketing program for free software. It's a pitch for "free software" on solid pragmatic grounds rather than ideological tub-thumping. The winning substance has not changed, the losing attitude and symbolism have. ...*
>
> *The case that needs to be made to most techies isn't about the concept of open source, but the name. Why not call it, as we traditionally have, free software?*
>
> *One direct reason is that the term "free software" is easily misunderstood in ways that lead to conflict. ...*
>
> *But the real reason for the re-labeling is a marketing one. We're trying to pitch our concept to the corporate world now. We have a winning product, but our positioning, in the past, has been awful. The term "free software" has been misunderstood by business persons, who mistake the desire to share with anti-commercialism, or worse, theft.*
>
> *Mainstream corporate CEOs and CTOs will never buy "free software." But if we take the very same tradition, the same people, and the same free-software licenses and change the label to "open source"* — that, *they'll buy.*
>
> *Some hackers find this hard to believe, but that's because they're techies who think in concrete, substantial terms and don't understand how important image is when you're selling something.*
>
> *In marketing, appearance is reality. The appearance that we're willing to climb down off the barricades and work with the corporate world counts for as much as the reality of our behavior, our convictions, and our software.*
>
> (from opensource.org [https://www.opensource.org/]. Or rather, *formerly* from that site — the OSI has apparently taken down the pages since then, although

[9] OSI's web home is opensource.org [https://www.opensource.org/].

> they can still be seen at web.archive.org/web/20021204155057/http://www.opensource.org/advocacy/faq.php [https://web.archive.org/web/20021204155057/http://www.opensource.org/advocacy/faq.php] and web.archive.org/web/20021204155022/http://www.opensource.org/advocacy/case_for_hackers.php#marketing [https://web.archive.org/web/20021204155022/http://www.opensource.org/advocacy/case_for_hackers.php#marketing] [sic].)

The tips of many icebergs of controversy are visible in that text. It refers to "our convictions", but smartly avoids spelling out exactly what those convictions are. For some, it might be the conviction that code developed according to an open process will be better code; for others, it might be the conviction that all information should be shared. There's the use of the word "theft" to refer (presumably) to illegal copying—a usage that many object to, on the grounds that it's not theft if the original possessor still has the item afterwards. There's the tantalizing hint that the free software movement might be mistakenly accused of anti-commercialism, but it leaves carefully unexamined the question of whether such an accusation would have any basis in fact.

None of which is to say that the OSI's web site is inconsistent or misleading. It's not. Rather, it is an example of exactly what the OSI claims had been missing from the free software movement: good marketing, where "good" means "viable in the business world." The Open Source Initiative gave a lot of people exactly what they had been looking for—a vocabulary for talking about free software as a development methodology and business strategy, instead of as a moral crusade.

The appearance of the Open Source Initiative changed the landscape of free software. It formalized a dichotomy that had long been unnamed, and in doing so forced the movement to acknowledge that it had internal politics as well as external. The effect today is that both sides have had to find common ground, since most projects include programmers from both camps, as well as participants who don't fit any clear category. This doesn't mean people never talk about moral motivations—lapses in the traditional "hacker ethic" are sometimes called out, for example. But it is rare for a free software / open source developer to openly question the basic motivations of others in a project. The contribution trumps the contributor. If someone writes good code, you don't ask them whether they do it for moral reasons, or because their employer paid them to, or because they're building up their résumé, or whatever. You evaluate the contribution on technical grounds, and respond on technical grounds. Even explicitly political organizations like the Debian project, whose goal is to offer a 100% free (that is, "free as in freedom") computing environment, are fairly relaxed about integrating with non-free code and cooperating with programmers who don't share exactly the same goals.

The Situation Today

When running a free software project, you won't need to talk about such weighty philosophical matters on a daily basis. Programmers will not insist that everyone else in the project agree with their views on all things (those who do insist on this quickly find themselves unable to work in any project). But you do need to be aware that the question of "free" versus "open source" exists, partly to avoid saying things that might be inimical to some of the participants, and partly because understanding developers' motivations is the best way—in some sense, the *only* way—to manage a project.

Free software is a culture by choice. To operate successfully in it, you have to understand why people choose to be in it in the first place. Coercive techniques don't work. If people are unhappy in one project, they will just wander off to another one. Free software is remarkable even among intentional communities for its lightness of investment. Many of the people involved have never actually met the other participants face-to-face. The normal conduits by which humans bond with each other and form lasting groups are narrowed down to a tiny channel: the written word, carried over electronic wires. Because of this, it can take a long time for a cohesive and dedicated group to form. Conversely, it's quite easy for a project to lose a potential participant in the first five minutes of acquaintanceship. If a project doesn't make a good first impression, newcomers may wait a long time before giving it a second chance.

The transience, or rather the *potential* transience, of relationships is perhaps the single most daunting task facing a new project. What will persuade all these people to stick together long enough to produce something useful? The answer to that question is complex enough to occupy the rest of this book, but if it had to be expressed in one sentence, it would be this:

> *People should feel that their connection to a project, and influence over it, is directly proportional to their contributions.*

No class of developers, or potential developers, should ever feel discounted or discriminated against for non-technical reasons[10]. Clearly, projects with corporate sponsorship and/or salaried developers need to be especially careful in this regard, as Chapter 5, *Organizations, Money, and Business* discusses in detail. Of course, this doesn't mean that if there's no corporate sponsorship then you have nothing to worry about. Money is merely one of many factors that can affect the success of a project. There are also questions of what language to choose, what license, what development process, precisely what kind of infrastructure to set up, how to publicize the project's inception effectively, and much more. Starting a project out on the right foot is the topic of the next chapter.

[10]There can be cases where you discriminate against certain developers due to behavior which, though not related to their technical contributions, has the potential to harm the project. That's reasonable: their behavior is relevant because in the long run it will have a negative effect on the project. The varieties of human culture being what they are, I can give no single, succint rule to cover all such cases, except to say that you should try to be welcoming to all potential contributors and, if you must discriminate, do so only on the basis of actual behavior, not on the basis of a contributor's group affiliation or group identity.

Chapter 2. Getting Started

The classic model of how free software projects get started was supplied by Eric Raymond, in a now-famous paper on open source processes entitled *The Cathedral and the Bazaar*. He wrote:

> *Every good work of software starts by scratching a developer's personal itch.*
>
> (from **catb.org/~esr/writings/cathedral-bazaar/ [http://www.catb.org/~esr/writings/cathedral-bazaar/]**)

Note that Raymond wasn't saying that open source projects happen only when some individual gets an itch. Rather, he was saying that *good* software results when the programmer has a personal interest in seeing the problem solved; the relevance of this to free software was that a personal itch happened to be the most frequent motivation for starting a free software project.

This is still how most free software projects are started, but less so now than in 1997, when Raymond wrote those words. Today, we have the phenomenon of organizations—for-profit corporations, governments, non-profits, etc—starting large, centrally-conceived open source projects from scratch. The lone programmer, banging out some code to solve a local problem and then realizing the result has wider applicability, is still the source of much new free software, but is not the only story.

Raymond's point is still insightful, however. The essential condition is that the producers of the software have a direct interest in its success, usually because they use it themselves or work directly with people who use it. If the software doesn't do what it's supposed to do, the person or organization producing it will feel the dissatisfaction in their daily work. For example, the open source software developed by the Kuali Foundation (kuali.org [https://www.kuali.org/]), used by educational institutions to manage their finances, research grants, HR systems, student information, etc, can hardly be said to scratch any individual programmer's personal itch. It scratches an institutional itch. But that itch arises directly from the experiences of the institutions concerned, and therefore if the project fails to satisfy them, they will know. This arrangement produces good software because the feedback loop flows in the right direction. The program isn't being written to be sold to someone else so they can solve *their* problem. It's being written to solve one's *own* problem, and then shared with everyone, much as though the problem were a disease and the software were medicine whose distribution is meant to completely eradicate the epidemic.

This chapter is about how to introduce a new free software project to the world, but many of its recommendations would sound familiar to a health organization distributing medicine. The goals are very similar: you want to make it clear what the medicine does, get it into the hands of the right people, and make sure that those who receive it know how to use it. But with software, you also want to entice some of the recipients into joining the ongoing research effort to improve the medicine.

Free software distribution is a twofold task. The software needs to acquire users, and to acquire developers. These two needs are not necessarily in conflict, but they do add some complexity to a project's initial presentation. Some information is useful for both audiences, some is useful only for one or the other. Both kinds of information should subscribe to the principle of scaled presentation; that is, the degree of detail presented at each stage should correspond to the amount of time and effort put in by the reader at that stage. More effort should always result in more reward. When the two do not correlate tightly, people may quickly lose faith and stop investing effort.

The corollary to this is that *appearances matter*. Programmers, in particular, often don't like to believe this. Their love of substance over form is almost a point of professional pride. It's no accident that so many programmers exhibit an antipathy for marketing and public relations work, nor that professional graphic designers are often horrified at the designs programmers come up with on their own.

This is a pity, because there are situations where form *is* substance, and project presentation is one of them. For example, the very first thing a visitor learns about a project is what its web site looks like. This information is absorbed before any of the actual content on the site is comprehended—before any of the text has been read or links clicked on. However unjust it may be, people cannot stop themselves from forming an immediate first impression. The site's appearance signals whether care was taken in organizing the project's presentation. Humans have extremely sensitive antennae for detecting the investment of care. Most of us can tell in one glance whether a web site was thrown together quickly or was given serious thought. This is the first piece of information your project puts out, and the impression it creates will carry over to the rest of the project by association.

Thus, while much of this chapter talks about the content your project should start out with, remember that its look and feel matter too. Because the project web site has to work for two different types of visitors—users and developers—special attention must be paid to clarity and directedness. Although this is not the place for a general treatise on web design, one principle is important enough to deserve mention, particularly when the site serves multiple (if overlapping) audiences: people should have a rough idea where a link goes before clicking on it. For example, it should be obvious *from looking at the links* to user documentation that they lead to user documentation, and not to, say, developer documentation. Running a project is partly about supplying information, but it's also about supplying comfort. The mere presence of certain standard offerings, in expected places, reassures users and developers who are deciding whether they want to get involved. It says that this project has its act together, has anticipated the questions people will ask, and has made an effort to answer them in a way that requires minimal exertion on the part of the asker. By giving off this aura of preparedness, the project sends out a message: "Your time will not be wasted if you get involved," which is exactly what people need to hear.

If you use a "canned hosting" site (see the section called “Hosting”), one advantage of that choice is that those sites have a default layout that is similar from project to project, and is pretty well-suited to presenting a project to the world. That layout can be customized, within certain boundaries, but the default design prompts you to include the information visitors are most likely to be looking for.

But First, Look Around

Before starting an open source project, there is one important caveat:

Always look around to see if there's an existing project that does what you want. The chances are pretty good that whatever problem you want solved now, someone else wanted solved before you. If they did solve it, and released their code under a free license, then there's no reason for you to reinvent the wheel today. There are exceptions, of course: if you want to start a project as an educational experience, pre-existing code won't help; or maybe the project you have in mind is so specialized that you know there is zero chance anyone else has done it. But generally, there's no point not looking, and the payoff can be huge. If the usual Internet search engines don't turn up anything, try searching directly on github.com [https://github.com/], freshcode.club [https://freshcode.club/], openhub.net [https://openhub.net/], sourceforge.net [https://sourceforge.net/], and in the Free Software Foundation's directory of free software at directory.fsf.org [https://directory.fsf.org/].

Even if you don't find exactly what you were looking for, you might find something so close that it makes more sense to join that project and add functionality than to start from scratch yourself. See the section called “Evaluating Open Source Projects”, in Chapter 5, *Organizations, Money, and Business* for a discussion of how to evaluate an existing open source project quickly.

Starting From What You Have

You've looked around, found that nothing out there really fits your needs, and decided to start a new project.

What now?

The hardest part about launching a free software project is transforming a private vision into a public one. You or your organization may know perfectly well what you want, but expressing that goal comprehensibly to the world is a fair amount of work. It is essential, however, that you take the time to do it. You and the other founders must decide what the project is really about—that is, decide its limitations, what it *won't* do as well as what it will—and write up a mission statement. This part is usually not too hard, though it can sometimes reveal unspoken assumptions and even disagreements about the nature of the project, which is fine: better to resolve those now than later. The next step is to package up the project for public consumption, and this is, basically, pure drudgery.

What makes it so laborious is that it consists mainly of organizing and documenting things everyone already knows—"everyone", that is, who's been involved in the project so far. Thus, for the people doing the work, there is no immediate benefit. They do not need a `README` file giving an overview of the project, nor a design document. They do not need a carefully arranged code tree conforming to the informal but widespread standards of software source distributions. Whatever way the source code is arranged is fine for them, because they're already accustomed to it anyway, and if the code runs at all, they know how to use it. It doesn't even matter, for them, if the fundamental architectural assumptions of the project remain undocumented; they're already familiar with that too.

Newcomers, on the other hand, need all these things. Fortunately, they don't need them all at once. It's not necessary for you to provide every possible resource before taking a project public. In a perfect world, perhaps, every new open source project would start out life with a thorough design document, a complete user manual (with special markings for features planned but not yet implemented), beautifully and portably packaged code, capable of running on any computing platform, and so on. In reality, taking care of all these loose ends would be prohibitively time-consuming, and anyway, it's work that one can reasonably hope others will help with once the project is under way.

What *is* necessary, however, is that enough investment be put into presentation that newcomers can get past the initial obstacle of unfamiliarity. Think of it as the first step in a bootstrapping process, to bring the project to a kind of minimum activation energy. I've heard this threshold called the *hacktivation energy*: the amount of energy a newcomer must put in before she starts getting something back. The lower a project's hacktivation energy, the better. Your first task is bring the hacktivation energy down to a level that encourages people to get involved.

Each of the following subsections describes one important aspect of starting a new project. They are presented roughly in the order that a new visitor would encounter them, though of course the order in which you actually implement them might be different. You can treat them as a checklist. When starting a project, just go down the list and make sure you've got each item covered, or at least that you're comfortable with the potential consequences if you've left one out.

Choose a Good Name

Put yourself in the shoes of someone who's just heard about your project, perhaps by having stumbled across it while searching for software to solve some problem. The first thing they'll encounter is the project's name.

A good name will not automatically make your project successful, and a bad name will not doom it—well, a *really* bad name probably could do that, but we start from the assumption that no one here is actively trying to make their project fail. However, a bad name can slow down adoption of the project, either because people don't take it seriously, or because they simply have trouble remembering it.

A good name:

- Gives some idea what the project does, or at least is related in an obvious way, such that if one knows the name and knows what the project does, the name will come quickly to mind thereafter.

- Is easy to remember. Here, there is no getting around the fact that English has become the default language of the Internet: "easy to remember" usually means "easy for someone who can read English to remember." Names that are puns dependent on native-speaker pronounciation, for example, will be opaque to the many non-native English readers out there. If the pun is particularly compelling and memorable, it may still be worth it; just keep in mind that many people seeing the name will not hear it in their head the way a native speaker would.

- Is not the same as some other project's name, and does not infringe on any trademarks. This is just good manners, as well as good legal sense. You don't want to create identity confusion. It's hard enough to keep track of everything that's available on the Net already, without different things having the same name.

  The resources mentioned earlier in the section called “But First, Look Around” are useful in discovering whether another project already has the name you're thinking of. For the U.S., trademark searches are available at uspto.gov [http://www.uspto.gov/].

- If possible, is available as a domain name in the `.com`, `.net`, and `.org` top-level domains. You should pick one, probably `.org`, to advertise as the official home site for the project; the other two should forward there and are simply to prevent third parties from creating identity confusion around the project's name. Even if you intend to host the project at some other site (see the section called “Hosting”), you can still register project-specific domains and forward them to the hosting site. It helps users a lot to have a simple URL to remember.

- If possible, is available as a username on Twitter [https://twitter.com/] and other microblog sites. See the section called “Own the name in the important namespaces” for more on this and its relationship to the domain name.

Own the name in the important namespaces

For large projects, it is a good idea to own the project's name as many of the relevant namespaces on the Internet as you can. By namespaces, I mean not just the domain name system, but also online services in which account names (usernames) are the publicly visible handle by which people refer to the project. If you have the same name in all the places where people would look for you, you make it easier for people to sustain a mild interest in the project until they're ready to become more involved.

For example, the Gnome free desktop project has the gnome.org [https://gnome.org/] domain name[1], the @gnome [https://twitter.com/gnome] Twitter handle, the gnome [https://identi.ca/gnome] username at Identi.ca[2], the gnome [https://github.com/gnome] username at GitHub.com[3], and on the freenode IRC network (see the section called “IRC / Real-Time Chat Systems”) they have the channel `#gnome`, although they also maintain their own IRC servers (where they control the channel namespace anyway, of course).

All this makes the Gnome project splendidly easy to find: it's usually right where a potential contributor would expect it to be. Of course, Gnome is a large and complex project with thousands of contributors and many subdivisions; the advantage to Gnome of being easy to find is greater than it would be for a newer project, since by now there are so many ways to get involved in Gnome. But it will certain-

[1] They didn't manage to get gnome.com or gnome.net, but that's okay — if you only have one, and it's .org, it's fine. That's usually the first one people look for when they're seeking the open source project of that name. If they couldn't get "gnome.org" itself, a typical solution would be to get "gnomeproject.org" instead, and many projects solve the problem that way.

[2] Identi.ca [https://identi.ca/] is a microblog / social networking that a number of free software developers use; its code is open source and made available at pump.io [http://pump.io/]. For developer-oriented projects, I recommend at least doing all status microposts — colloquially referred to as "tweets" — on both Identi.ca and Twitter. While the total number of people on Identi.ca is far smaller than on Twitter, the percentage of them that are likely to be interested in news about an open source project is far higher, at least as of this writing in 2013 and for some years preceding that.

[3] While the master copy of Gnome's source code is at git.gnome.org [https://git.gnome.org/], they maintain a mirror at GitHub, since so many developers are already familiar with GitHub

ly never *harm* your project to own its name in as many of the relevant namespaces as it can, and it can sometimes help. So when you start a project, think about what its online handle should be and register that handle with the online services you think you're likely to care about. The ones mentioned above are probably a good initial list, but you may know others that are relevant for the particular subject area of your project.

Have a Clear Mission Statement

Once they've found the project's home site, the next thing people will look for is a quick description or mission statement, so they can decide (within 30 seconds) whether or not they're interested in learning more. This should be prominently placed on the front page, preferably right under the project's name.

The description should be concrete, limiting, and above all, short. Here's an example of a good one, from hadoop.apache.org [https://hadoop.apache.org/]:

> *The Apache™ Hadoop® project develops open-source software for reliable, scalable, distributed computing.*
>
> *The Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage. Rather than rely on hardware to deliver high-availability, the library itself is designed to detect and handle failures at the application layer, so delivering a highly-available service on top of a cluster of computers, each of which may be prone to failures.*

In just four sentences, they've hit all the high points, largely by drawing on the reader's prior knowledge. That's an important point: it's okay to assume a minimally informed reader with a baseline level of preparedness. A reader who doesn't know what "clusters" and "high-availability" mean in this context probably can't make much use of Hadoop anyway, so there's no point writing for a reader who knows any less than that. The phrase "designed to detect and handle failures at the application layer" will stand out to engineers who have experience with large-scale computing clusters—when they see those words, they'll know that the people behind Hadoop understand that world, and will thus be more willing to give Hadoop consideration.

Those who remain interested after reading the mission statement will next want to see more details, perhaps some user or developer documentation, and eventually will want to download something. But before any of that, they'll need to be sure it's open source.

State That the Project is Free

The front page must make it unambiguously clear that the project is open source. This may seem obvious, but you would be surprised how many projects forget to do it. I have seen free software project web sites where the front page not only did not say which particular free license the software was distributed under, but did not even state outright that the software was free at all. Sometimes the crucial bit of information was relegated to the Downloads page, or the Developers page, or some other place that required one more mouse click to get to. In extreme cases, the license was not given anywhere on the web site at all—the only way to find it was to download the software and look at a license file inside.

Please don't make this mistake. Such an omission can lose many potential developers and users. State up front, right below the mission statement, that the project is "free software" or "open source software", and give the exact license. A quick guide to choosing a license is given in the section called “Choosing a License and Applying It” later in this chapter, and licensing issues are discussed in detail in Chapter 9, *Legal Matters: Licenses, Copyrights, Trademarks and Patents*.

By this point, our hypothetical visitor has determined—probably in a minute or less—that she's interested in spending, say, at least five more minutes investigating this project. The next sections describe what she should encounter in that five minutes.

Features and Requirements List

There should be a brief list of the features the software supports (if something isn't completed yet, you can still list it, but put "*planned*" or "*in progress*" next to it), and the kind of computing environment required to run the software. Think of the features/requirements list as what you would give to someone asking for a quick summary of the software. It is often just a logical expansion of the mission statement. For example, the mission statement might say:

> *To create a full-text indexer and search engine with a rich API, for use by programmers in providing search services for large collections of text files.*

The features and requirements list would give the details, clarifying the mission statement's scope:

> *Features:*
>
> - *Searches plain text, HTML, and XML*
> - *Word or phrase searching*
> - *(planned) Fuzzy matching*
> - *(planned) Incremental updating of indexes*
> - *(planned) Indexing of remote web sites*
>
> *Requirements:*
>
> - *Python 2.2 or higher*
> - *Enough disk space to hold the indexes (approximately 2x original data size)*

With this information, readers can quickly get a feel for whether this software has any hope of working for them, and they can consider getting involved as developers too.

Development Status

Visitors usually want to know how a project is doing. For new projects, they want to know the gap between the project's promise and current reality. For mature projects, they want to know how actively it is maintained, how often it puts out new releases, how responsive it is likely to be to bug reports, etc.

There are a couple of different avenues for providing answers to these questions. One is to have a development status page, listing the project's near-term goals and needs (for example, it might be looking for developers with a particular kind of expertise). The page can also give a history of past releases, with feature lists, so visitors can get an idea of how the project defines "progress", and how quickly it makes progress according to that definition. Some projects structure their development status page as a roadmap that includes the future: past events are shown on the dates they actually happened, future ones on the approximate dates the project hopes they will happen.

The other way — not mutually exclusive with the first, and in fact probably best done in combination with it — is to have various automatically-maintained counters and indicators embedded in the projects's front page and/or its developer landing page, showing various pieces of information that, in the aggregate, give a sense of the project's development status and progress. For example, an Announcements or News panel showing recent news items, a Twitter or other microblog stream showing notices that match

the project's designated hashtags, a timeline of recent releases, a panel showing recent activity in the bug tracker (bugs filed, bugs responded to), another showing mailing list or discussion forum activity, etc. Each such indicator should be a gateway to further information of its type: for example, clicking on the "recent bugs" panel should take one to the full bug tracker, or at least to an expanded view into bug tracker activity.

Really, there are two slightly different meanings of "development status" being conflated here. One is the formal sense: where does the project stand in relation to its stated goals, and how fast is it making progress. The other is less formal but just as useful: how active is this project? Is stuff going on? Are there people here, getting things done? Often that latter notion is what a visitor is most interested in. Whether or not a project met its most recent milestone is sometimes not as interesting as the more fundamental question of whether it has an active community of developers around it.

The two notions of development status are, of course, related, and a well-presented project shows both kinds. The information can be divided between the project's front page (show enough there to give an overview of both types of development status) and a more developer-oriented page.

Example: Launchpad Status Indicators

One site that does a pretty good job of showing developer-oriented status indicators is Launchpad.net. Launchpad.net is a bit unusual in that it is both a primary hosting platform for some projects, and a secondary, packaging-oriented site for others (or rather, for those others it is the primary site for the "project" of getting that particular program packaged for the Ubuntu GNU/Linux operating system, which Launchpad was specifically designed to support). In either case, a project's landing page on Launchpad shows a variety of automatically-maintained status indicators that quickly give an idea of where the project stands. While simply imitating a Launchpad page is probably not a good idea — your own project should think carefully about what its best development status indicators are — Launchpad project pages do provide some good examples of the possibilities. Start from the top of a project page there and scroll down: launchpad.net/drizzle [https://launchpad.net/drizzle]. or launchpad.net/inkscape [https://launchpad.net/inkscape], to pick two at random.

Development status should always reflect reality.

Don't be afraid of looking unready, and never give in to the temptation to inflate or hype the development status. Everyone knows that software evolves by stages; there's no shame in saying "This is alpha software with known bugs. It runs, and works at least some of the time, but use at your own risk." Such language won't scare away the kinds of developers you need at that stage. As for users, one of the worst things a project can do is attract users before the software is ready for them. A reputation for instability or bugginess is very hard to shake, once acquired. Conservativism pays off in the long run; it's always better for the software to be *more* stable than the user expected than less, and pleasant surprises produce the best kind of word-of-mouth.

Alpha and Beta

The term *alpha* usually means a first release, with which users can get real work done and which has all the intended functionality, but which also has known bugs. The main purpose of alpha software is to generate feedback, so the developers know what to work on. The next stage, *beta*, means the software has had all the serious bugs fixed, but has not yet been tested enough to certify for production release. The purpose of beta software is to either become the official release, assuming no bugs are found, or provide detailed feedback to the developers so they can reach the official release quickly. The difference between alpha and beta is very much a matter of judgement.

Downloads

The software should be downloadable as source code in standard formats. When a project is first getting started, binary (executable) packages are not necessary, unless the software has such complicated build requirements or dependencies that merely getting it to run would be a lot of work for most people. (But if this is the case, the project is going to have a hard time attracting developers anyway!)

The distribution mechanism should be as convenient, standard, and low-overhead as possible. If you were trying to eradicate a disease, you wouldn't distribute the medicine in such a way that it requires a non-standard syringe size to administer. Likewise, software should conform to standard build and installation methods; the more it deviates from the standards, the more potential users and developers will give up and go away confused.

That sounds obvious, but many projects don't bother to standardize their installation procedures until very late in the game, telling themselves they can do it any time: *"We'll sort all that stuff out when the code is closer to being ready."* What they don't realize is that by putting off the boring work of finishing the build and installation procedures, they are actually making the code take longer to get ready —because they discourage developers who might otherwise have contributed to the code, if only they could build and test it. Most insidiously, the project won't even *know* it's losing all those developers, because the process is an accumulation of non-events: someone visits a web site, downloads the software, tries to build it, fails, gives up and goes away. Who will ever know it happened, except the person themselves? No one working on the project will realize that someone's interest and good will have been silently squandered.

Boring work with a high payoff should always be done early, and significantly lowering the project's barrier to entry through good packaging brings a very high payoff.

When you release a downloadable package, give it a unique version number, so that people can compare any two releases and know which supersedes the other. That way they can report bugs against a particular release (which helps respondents to figure out if the bug is already fixed or not). A detailed discussion of version numbering can be found in the section called “Release Numbering”, and the details of standardizing build and installation procedures are covered in the section called “Packaging”, both in Chapter 7, *Packaging, Releasing, and Daily Development*.

Version Control and Bug Tracker Access

Downloading source packages is fine for those who just want to install and use the software, but it's not enough for those who want to debug or add new features. Nightly source snapshots can help, but they're still not fine-grained enough for a thriving development community. People need real-time access to the latest sources, and a way to submit changes based on those sources.

The solution is to use a version control system — specifically, an online, publicly-accessible version controlled repository, from which anyone can check out the project's materials and subsequently get updates. A version control repository is a sign—to both users and developers—that this project is making an effort to give people what they need to participate. As of this writing, many open source projects use GitHub.com [https://github.com/], which offers unlimited free public version control hosting for open source projects. While GitHub is not the only choice, nor even the only good choice, it's a reasonable one for most projects[4]. Version control infrastructure is discussed in detail in the section called “Version Control” in Chapter 3, *Technical Infrastructure*.

The same goes for the project's bug tracker. The importance of a bug tracking system lies not only in its day-to-day usefulness to developers, but in what it signifies for project observers. For many people,

[4]Although GitHub is based on Git, a popular open source version control system, the code that runs GitHub's web services is not itself open source. Whether this matters for your project is a complex question, and is addressed in more depth in the section called “Canned Hosting” in Chapter 3, *Technical Infrastructure*

an accessible bug database is one of the strongest signs that a project should be taken seriously — and the higher the number of bugs in the database, the *better* the project looks. That might seem counterintuitive, but remember that the number of bug reports filed really depends on three things: the absolute number of actual software defects present in the code, the number of people using the software, and the convenience with which those people can report new bugs. Of these three factors, the latter two are much more significant than the first. Any software of sufficient size and complexity has an essentially arbitrary number of bugs waiting to be discovered. The real question is, how well will the project do at recording and prioritizing those bugs? A project with a large and well-maintained bug database (meaning bugs are responded to promptly, duplicate bugs are unified, etc.) therefore makes a better impression than a project with no bug database, or a nearly empty database.

Of course, if your project is just getting started, then the bug database will contain very few bugs, and there's not much you can do about that. But if the status page emphasizes the project's youth, and if people looking at the bug database can see that most filings have taken place recently, they can extrapolate from that the project still has a healthy *rate* of filings, and they will not be unduly alarmed by the low absolute number of bugs recorded.[5]

Note that bug trackers are often used to track not only software bugs, but enhancement requests, documentation changes, pending tasks, and more. The details of running a bug tracker are covered in the section called “Bug Tracker” in Chapter 3, *Technical Infrastructure*, so I won't go into them here. The important thing from a presentation point of view is just to *have* a bug tracker, and to make sure that fact is visible from the front page of the project.

Communications Channels

Visitors usually want to know how to reach the human beings involved with the project. Provide the addresses of mailing lists, chat rooms, IRC channels (Chapter 3, *Technical Infrastructure*), and any other forums where others involved with the software can be reached. Make it clear that you and the other authors of the project are subscribed to these mailing lists, so people see there's a way to give feedback that will reach the developers. Your presence on the lists does not imply a committment to answer all questions or implement all feature requests. In the long run, probably only a fraction users will use the forums anyway, but the others will be comforted to know that they *could* if they ever needed to.

In the early stages of a project, there's no need to have separate user and developer forums. It's much better to have everyone involved with the software talking together, in one "room." Among early adopters, the distinction between developer and user is often fuzzy; to the extent that the distinction can be made, the ratio of developers to users is usually much higher in the early days of the project than later on. While you can't assume that every early adopter is a programmer who wants to hack on the software, you can assume that they are at least interested in following development discussions and in getting a sense of the project's direction.

As this chapter is only about getting a project started, it's enough merely to say that these communications forums need to exist. Later, in the section called “Handling Growth” in Chapter 6, *Communications*, we'll examine where and how to set up such forums, the ways in which they might need moderation or other management, and how to separate user forums from developer forums, when the time comes, without creating an unbridgeable gulf.

[5]For a more thorough argument that bug reports should be treated as good news, see rants.org/2010/01/10/bugs-users-and-tech-debt [http://www.rants.org/2010/01/10/bugs-users-and-tech-debt/], an article I wrote in 2010 about how bug reports do *not* represent "technical debt [https://en.wikipedia.org/wiki/Technical_debt]" but rather user engagement.

Developer Guidelines

If someone is considering contributing to the project, she'll look for developer guidelines. Developer guidelines are not so much technical as social: they explain how the developers interact with each other and with the users, and ultimately how things get done.

This topic is covered in detail in the section called “Writing It All Down” in Chapter 4, *Social and Political Infrastructure*, but the basic elements of developer guidelines are:

- pointers to forums for interaction with other developers
- instructions on how to report bugs and submit patches
- some indication of *how* development is usually done and how decisions are made—is the project a benevolent dictatorship, a democracy, or something else

No pejorative sense is intended by "dictatorship", by the way. It's perfectly okay to run a tyranny where one particular developer has veto power over all changes. Many successful projects work this way. The important thing is that the project come right out and say so. A tyranny pretending to be a democracy will turn people off; a tyranny that says it's a tyranny will do fine as long as the tyrant is competent and trusted. (See the section called “Forkability” in Chapter 4, *Social and Political Infrastructure* for why dictatorship in open source projects doesn't have the same implications as dictatorship in other areas of life.)

subversion.apache.org/docs/community-guide [http://subversion.apache.org/docs/community-guide/] is an example of particularly thorough developer guidelines; the LibreOffice guidelines at wiki.documentfoundation.org/Development [https://wiki.documentfoundation.org/Development] are also a good example.

The separate issue of providing a programmer's introduction to the software is discussed in the section called “Developer documentation” later in this chapter.

Documentation

Documentation is essential. There needs to be *something* for people to read, even if it's rudimentary and incomplete. This falls squarely into the "drudgery" category referred to earlier, and is often the first area where a new open source project falls down. Coming up with a mission statement and feature list, choosing a license, summarizing development status—these are all relatively small tasks, which can be definitively completed and usually need not be revisited once done. Documentation, on the other hand, is never really finished, which may be one reason people sometimes delay starting it at all.

The most insidious thing is that documentation's utility to those writing it is the reverse of its utility to those who will read it. The most important documentation for initial users is the basics: how to quickly set up the software, an overview of how it works, perhaps some guides to doing common tasks. Yet these are exactly the things the *writers* of the documentation know all too well—so well that it can be difficult for them to see things from the reader's point of view, and to laboriously spell out the steps that (to the writers) seem so obvious as to be unworthy of mention.

There's no magic solution to this problem. Someone just needs to sit down and write the stuff, and then, most importantly, incorporate feedback from readers. Use a simple, easy-to-edit format such as HTML, plain text, Markdown, ReStructuredText, or some variant of XML—something that's convenient for lightweight, quick improvements on the spur of the moment[6]. This is not only to remove any overhead

[6]Don't worry too much about choosing the right format the first time. If you change your mind later, you can always do an automated conversion using Pandoc [http://johnmacfarlane.net/pandoc/].

that might impede the original writers from making incremental improvements, but also for those who join the project later and want to work on the documentation.

One way to ensure basic initial documentation gets done is to limit its scope in advance. That way, writing it at least won't feel like an open-ended task. A good rule of thumb is that it should meet the following minimal criteria:

- Tell the reader clearly how much technical expertise they're expected to have.

- Describe clearly and thoroughly how to set up the software, and somewhere near the beginning of the documentation, tell the user how to run some sort of diagnostic test or simple command to confirm that they've set things up correctly. Startup documentation is in some ways more important than actual usage documentation. The more effort someone has invested in installing and getting started with the software, the more persistent she'll be in figuring out advanced functionality that's not well-documented. When people abandon, they abandon early; therefore, it's the earliest stages, like installation, that need the most support.

- Give one tutorial-style example of how to do a common task. Obviously, many examples for many tasks would be even better, but if time is limited, pick one task and walk through it thoroughly. Once someone sees that the software *can* be used for one thing, they'll start to explore what else it can do on their own—and, if you're lucky, start filling in the documentation themselves. Which brings us to the next point...

- Label the areas where the documentation is known to be incomplete. By showing the readers that you are aware of its deficiencies, you align yourself with their point of view. Your empathy reassures them that they don't face a struggle to convince the project of what's important. These labels needn't represent promises to fill in the gaps by any particular date —it's equally legitimate to treat them as open requests for help.

The last point is of wider importance, actually, and can be applied to the entire project, not just the documentation. An accurate accounting of known deficiencies is the norm in the open source world. You don't have to exaggerate the project's shortcomings, just identify them scrupulously and dispassionately when the context calls for it (whether in the documentation, in the bug tracking database, or on a mailing list discussion). No one will treat this as defeatism on the part of the project, nor as a commitment to solve the problems by a certain date, unless the project makes such a commitment explicitly. Since anyone who uses the software will discover the deficiencies for themselves, it's much better for them to be psychologically prepared—then the project will look like it has a solid knowledge of how it's doing.

Maintaining a FAQ

A *FAQ* ("Frequently Asked Questions" document) can be one of the best investments a project makes in terms of educational payoff. FAQs are highly tuned to the questions users and developers actually ask—as opposed to the questions you might have *expected* them to ask—and therefore, a well-maintained FAQ tends to give those who consult it exactly what they're looking for. The FAQ is often the first place users look when they encounter a problem, often even in preference to the official manual, and it's probably the document in your project most likely to be linked to from other sites.

Unfortunately, you cannot make the FAQ at the start of the project. Good FAQs are not written, they are grown. They are by definition reactive documents, evolving over time in response to the questions people ask about the software. Since it's impossible to correctly anticipate those questions, it is impossible to sit down and write a useful FAQ from scratch.

Therefore, don't waste your time trying to. You may, however, find it useful to set up a mostly blank FAQ template with just a few questions and answers, so there will be an obvious place for people to contribute questions and answers after the project is under way. At this stage, the most important property is not completeness, but *convenience*: if the FAQ is easy to add to, people will add to it. (Proper FAQ maintenance is a non-trivial and intriguing problem: see the section called “"Manager" Does Not Mean "Owner"” in Chapter 8, *Managing Participants*, the section called “Q&A Forums” in Chapter 3, *Technical Infrastructure*, and the section called “Treat all resources like archives” in Chapter 6, *Communications*.)

Availability of documentation

Documentation should be available from two places: online (directly from the web site), *and* in the downloadable distribution of the software (see the section called “Packaging” in Chapter 7, *Packaging, Releasing, and Daily Development*). It needs to be online, in browsable form, because people often read documentation *before* downloading software for the first time, as a way of helping them decide whether to download at all. But it should also accompany the software, on the principle that downloading should supply (i.e., make locally accessible) everything one needs to use the package.

For online documentation, make sure that there is a link that brings up the *entire* documentation in one HTML page (put a note like "monolithic" or "all-in-one" or "single large page" next to the link, so people know that it might take a while to load). This is useful because people often want to search for a specific word or phrase across the entire documentation. Generally, they already know what they're looking for; they just can't remember what section it's in. For such people, nothing is more frustrating than encountering one HTML page for the table of contents, then a different page for the introduction, then a different page for installation instructions, etc. When the pages are broken up like that, their browser's search function is useless. The separate-page style is useful for those who already know what section they need, or who want to read the entire documentation from front to back in sequence. But this is not necessarily the most common way documentation is accessed. Often, someone who is basically familiar with the software is coming back to search for a specific word or phrase, and to fail to provide them with a single, searchable document would only make their lives harder.

Developer documentation

Developer documentation is written by programmers to help other programmers understand the code, so they can repair and extend it. This is somewhat different from the *developer guidelines* discussed earlier, which are more social than technical. Developer guidelines tell programmers how to get along with each other; developer documentation tells them how to get along with the code itself. The two are often packaged together in one document for convenience (as with the subversion.apache.org/docs/commu-

nity-guide [http://subversion.apache.org/docs/community-guide/] example given earlier), but they don't have to be.

Although developer documentation can be very helpful, there's no reason to delay a release to do it. As long as the original authors are available (and willing) to answer questions about the code, that's enough to start with. In fact, having to answer the same questions over and over is a common motivation for writing documentation. But even before it's written, determined contributors will still manage to find their way around the code. The force that drives people to spend time learning a codebase is that the code does something useful for them. If people have faith in that, they will take the time to figure things out; if they don't have that faith, no amount of developer documentation will get or keep them.

So if you have time to write documentation for only one audience, write it for users. All user documentation is, in effect, developer documentation as well; any programmer who's going to work on a piece of software will need to be familiar with how to use it too. Later, when you see programmers asking the same questions over and over, take the time to write up some separate documents just for them.

Some projects use wikis for their initial documentation, or even as their primary documentation. In my experience, this works best if the wiki is actively maintained by a few people who agree on how the documentation is to be organized and what sort of "voice" it should have. See the section called “Wikis” in Chapter 3, *Technical Infrastructure* for more.

Demos, Screenshots, Videos, and Example Output

If the project involves a graphical user interface, or if it produces graphical or otherwise distinctive output, put some samples up on the project web site. In the case of interface, this means screenshots or, better yet, a brief (4 minutes or fewer) video with subtitles or a narrator. For output, it might be screenshots or just sample files to download. For web-based software, the gold standard is a demo site, of course, assuming the software is amenable to that.

The main thing is to cater to people's desire for instant gratification in the way they are most likely to expect. A single screenshot or video can be more convincing than paragraphs of descriptive text and mailing list chatter, because it is proof that the software *works*. The code may still be buggy, it may be hard to install, it may be incompletely documented, but image-based evidence shows people that if one puts in enough effort, one can get it to run.

Keep Videos Brief, and *Say* They're Brief

If you have a video demonstration of your project, keep the video under 4 minutes long, and make sure people can see the duration *before* they click on it. This is in keeping with the "principle of scaled presentation" mentioned earlier: you want to make the decision to watch the video an easy one, by removing all the risk. Visitors are more likely to click on a link that says "Watch our 3 minute video" than on one that just says "Watch our video", because in the former case they know what they're getting into before they click — and they'll watch it better, because they've mentally prepared the necessary amount of commitment beforehand, and so won't tire mid-way through.

As to where the four-minute limit came from: it's a scientific fact, determined through many attempts by the same experimental subject (who shall remain unnamed) to watch project videos. The limit does not apply to tutorials or other instructional material, of course; it's just for introductory videos.

In case you don't already have preferred software for recording desktop interaction videos: I've had good luck with `gtk-recordmydesktop` on Debian GNU/Linux, and then the `OpenShot` video editor for post-capture editing.

There are many other things you could put on the project web site, if you have the time, or if for one reason or another they are especially appropriate: a news page, a project history page, a related links page, a site-search feature, a donations link, etc. None of these are necessities at startup time, but keep them in mind for the future.

Hosting

Where on the Internet should you put the project's materials?

A web site, obviously — but the full answer is a little more complicated than that.

Many projects distinguish between their primary public user-facing web site — the one with the pretty pictures and the "About" page and the gentle introductions and videos and guided tours and all that stuff — and their developers' site, where everything's grungy and full of closely-spaced text in monospace fonts and impenetrable abbreviations.

Well, I exaggerate. A bit. In any case, in the early stages of your project it is not so important to distinguish between these two audiences. Most of the interested visitors you get will be developers, or at least people who are comfortable trying out new code. Over time, you may find it makes sense to have a user-facing site (of course, if your project is a code library, those "users" might be other programmers) and a somewhat separate collaboration area for those interested in participating in development. The collaboration site would have the code repository, bug tracker, development wiki, links to development mailing lists, etc. The two sites should link to each other, and in particular it's important that the user-facing site make it clear that the project is open source and where the open source development activity can be found.[7]

In the past, many projects set up the developer site and infrastructure themselves. Over the last decade or so, however, most open source projects — and almost all the new ones — just use one of the "canned hosting" sites that have sprung up to offer these services for free to open source projects. By far the most popular such site, as of this writing in mid-2013, is GitHub.com [https://github.com/], and if you don't have a strong preference about where to host, you should probably just choose GitHub; many developers are already familiar with it and have personal accounts there. the section called “Canned Hosting” in Chapter 3, *Technical Infrastructure* has a more detailed discussion of the questions to consider when choosing a canned hosting site, and an overview of the most popular ones.

Choosing a License and Applying It

This section is intended to be a very quick, very rough guide to choosing a license. Read Chapter 9, *Legal Matters: Licenses, Copyrights, Trademarks and Patents* to understand the detailed legal implications of the different licenses, and how the license you choose can affect people's ability to mix your software with other software.

[7] As of August 2013, a good example of a project with separate but cross-linked primary and developer sites is the Ozone Widget Framework: compare their main user-facing site at ozoneplatform.org [http://www.ozoneplatform.org/] with their development area at github.com/ozoneplatform/owf [https://github.com/ozoneplatform/owf].

Synonyms: "free software license", "FSF-approved", "open source license", and "OSI-approved"

The terms "free software license" and "open source license" are essentially synonymous, and I treat them so throughout this book.

Technically, the former term refers to licenses confirmed by the Free Software Foundation as offering the "four freedoms" necessary for free software (see gnu.org/philosophy/free-sw.html [https://www.gnu.org/philosophy/free-sw.html]), while the latter term refers to licenses approved by the Open Source Initiative as meeting the Open Source Definition (opensource.org/osd [https://opensource.org/osd]). However, if you read the FSF's definition of free software, and the OSI's definition of open source software, it becomes obvious that the two definitions delineate the same freedoms — not surprisingly, as the section called “"Free" Versus "Open Source"” in Chapter 1, *Introduction* explains. The inevitable, and in some sense deliberate, result is that the two organizations have approved the same set of licenses.[8]

There are a great many free software licenses to choose from. Most of them we needn't consider here, as they were written to satisfy the particular legal needs of some corporation or person, and wouldn't be appropriate for your project. We will restrict ourselves to just the most commonly used licenses; in most cases, you will want to choose one of them.

The "Do Anything" Licenses

If you're comfortable with your project's code potentially being used in proprietary programs, then use an *MIT-style* license. It is the simplest of several minimal licenses that do little more than assert nominal copyright (without actually restricting copying) and specify that the code comes with no warranty. See the section called “Choosing a License” for details.

The GPL

If you don't want your code to be used in proprietary programs, use the GNU General Public License, version 3 (gnu.org/licenses/gpl.html [https://www.gnu.org/licenses/gpl.html]). The GPL is probably the most widely recognized free software license in the world today. This is in itself a big advantage, since many potential users and contributors will already be familiar with it, and therefore won't have to spend extra time to read and understand your license. See the section called “The GNU General Public License” in Chapter 9, *Legal Matters: Licenses, Copyrights, Trademarks and Patents* for details.

If users interact with your code primarily over a network—that is, the software is usually part of a hosted service, rather than being distributed as a binary—then consider using the *GNU Affero GPL* instead. The AGPL is just the GPL with one extra clause establishing network accessibility as a form of distribution for the purposes of the license. See the section called “The GNU Affero GPL: A Version of the GNU GPL for Server-Side Code” in Chapter 9, *Legal Matters: Licenses, Copyrights, Trademarks and Patents* for more.

How to Apply a License to Your Software

Once you've chosen a license, you'll need to apply it to the software.

[8] There are actually some minor differences between the sets of approved licenses, but they are not significant for our purposes — or indeed for most practical purposes. In some cases, one or the other organization has simply not gotten around to considering a given license, usually a license that is not widely-used anyway. And apparently (so I'm told) there historically was a license that at least one of the organizations, and possibly both, agreed fit one definition but not the other. Whenever I try to get the details on this, though, I seem to get a different answer as to what that license was, except that the license named is always one that was not many people used anyway. So today, for any license you are likely to be using, the terms "OSI-approved" and "FSF-approved" can be treated as implying each other.

The first thing to do is state the license clearly on the project's front page. You don't need to include the actual text of the license there; just give its name and make it link to the full license text on another page. That tells the public what license you *intend* the software to be released under—but it's not quite sufficient for legal purposes. The other step is that the software itself should include the license.

The standard way to do this is to put the full license text in a file called `COPYING` (or `LICENSE`) included with the source code, and then put a short notice in a comment at the top of each source file, naming the copyright date, holder, and license, and saying where to find the full text of the license.

There are many variations on this pattern, so we'll look at just one example here. The GNU GPL says to put a notice like this at the top of each source file:

```
Copyright (C) <year>  <name of author>

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see <http://www.gnu.org/licenses/>
```

It does not say specifically that the copy of the license you received along with the program is in the file `COPYING` or `LICENSE`, but that's where it's usually put. (You could change the above notice to state that directly, but there's no real need to.)

In general, the notice you put in each source file does not have to look exactly like the one above, as long as it starts with the same notice of copyright holder and date[9], states the name of the license, and makes clear where to view the full license terms. It's always best to consult a lawyer, of course, if you can afford one.

Setting the Tone

So far we've covered one-time tasks you do during project setup: picking a license, arranging the initial web site, etc. But the most important aspects of starting a new project are dynamic. Choosing a mailing list address is easy; ensuring that the list's conversations remain on-topic and productive is another matter entirely. For example, if the project is being opened up after years of closed, in-house development, its development processes will change, and you will have to prepare the existing developers for that change.

The first steps are the hardest, because precedents and expectations for future conduct have not yet been set. Stability in a project does not come from formal policies, but from a shared, hard-to-pin-down collective wisdom that develops over time. There are often written rules as well, but they tend to be essentially a distillation of the intangible, ever-evolving agreements that really guide the project. The written policies do not define the project's culture so much as describe it, and even then only approximately.

[9]The date should show the dates the file was modified, for copyright purposes. In other words, for a file modified in 2008, 2009, and 2013, you would write "2008, 2009, 2013" — *not* "2008-2013", because the file wasn't modified in most of the years in that range.

There are a few reasons why things work out this way. Growth and high turnover are not as damaging to the accumulation of social norms as one might think. As long as change does not happen *too* quickly, there is time for new arrivals to learn how things are done, and after they learn, they will help reinforce those ways themselves. Consider how children's songs survive for centuries. There are children today singing roughly the same rhymes as children did hundreds of years ago, even though there are no children alive now who were alive then. Younger children hear the songs sung by older ones, and when they are older, they in turn will sing them in front of other younger ones. The children are not engaging in a conscious program of transmission, of course, but the reason the songs survive is nonetheless that they are transmitted regularly and repeatedly. The time scale of free software projects may not be measured in centuries (we don't know yet), but the dynamics of transmission are much the same. The turnover rate is faster, however, and must be compensated for by a more active and deliberate transmission effort.

This effort is aided by the fact that people generally show up expecting and looking for social norms. That's just how humans are built. In any group unified by a common endeavor, people who join instinctively search for behaviors that will mark them as part of the group. The goal of setting precedents early is to make those "in-group" behaviors be ones that are useful to the project; once established, they will be largely self-perpetuating.

Following are some examples of specific things you can do to set good precedents. They're not meant as an exhaustive list, just as illustrations of the idea that setting a collaborative mood early helps a project tremendously. Physically, every developer may be working alone in a room by themselves, but you can do a lot to make them *feel* like they're all working together in the same room. The more they feel this way, the more time they'll want to spend on the project. I chose these particular examples because they came up in the Subversion project (subversion.apache.org [http://subversion.apache.org/]), which I participated in and observed from its very beginning. But they're not unique to Subversion; situations like these will come up in most open source projects, and should be seen as opportunities to start things off on the right foot.

Avoid Private Discussions

Even after you've taken the project public, you and the other founders will often find yourselves wanting to settle difficult questions by private communications among an inner circle. This is especially true in the early days of the project, when there are so many important decisions to make, and, usually, few people qualified to make them. All the obvious disadvantages of public list discussions will loom palpably in front of you: the delay inherent in email conversations, the need to leave sufficient time for consensus to form, the hassle of dealing with naive newcomers who think they understand all the issues but actually don't (every project has these; sometimes they're next year's star contributors, sometimes they stay naive forever), the person who can't understand why you only want to solve problem X when it's obviously a subset of larger problem Y, and so on. The temptation to make decisions behind closed doors and present them as *faits accomplis*, or at least as the firm recommendations of a united and influential voting block, will be great indeed.

Don't do it.

As slow and cumbersome as public discussion can be, it's almost always preferable in the long run. Making important decisions in private is like spraying contributor repellant on your project. No serious contributor would stick around for long in an environment where a secret council makes all the big decisions. Furthermore, public discussion has beneficial side effects that will last beyond whatever ephemeral technical question was at issue:

- The discussion will help train and educate new developers. You never know how many eyes are watching the conversation; even if most people don't participate, many may be lurking silently, gleaning information about the software.

- The discussion will train *you* in the art of explaining technical issues to people who are not as familiar with the software as you are. This is a skill that requires practice, and you can't get that practice by talking to people who already know what you know.

- The discussion and its conclusions will be available in public archives forever after, enabling future discussions to avoid retracing the same steps. See the section called “Conspicuous Use of Archives” in Chapter 6, *Communications*.

Finally, there is the possibility that someone on the list may make a real contribution to the conversation, by coming up with an idea you never anticipated. It's hard to say how likely this is; it just depends on the complexity of the code and degree of specialization required. But if anecdotal evidence may be permitted, I would hazard that this is more likely than you might intuitively expect. In the Subversion project, we (the founders) believed we faced a deep and complex set of problems, which we had been thinking about hard for several months, and we frankly doubted that anyone on the newly created mailing list was likely to make a real contribution to the discussion. So we took the lazy route and started batting some technical ideas back and forth in private emails, until an observer of the project[10] caught wind of what was happening and asked for the discussion to be moved to the public list. Rolling our eyes a bit, we did—and were stunned by the number of insightful comments and suggestions that quickly resulted. In many cases people offered ideas that had never even occurred to us. It turned out there were some *very* smart people on that list; they'd just been waiting for the right bait. It's true that the ensuing discussions took longer than they would have if we had kept the conversation private, but they were so much more productive that it was well worth the extra time.

Without descending into hand-waving generalizations like "the group is always smarter than the individual" (we've all met enough groups to know better), it must be acknowledged that there are certain activities at which groups excel. Massive peer review is one of them; generating large numbers of ideas quickly is another. The quality of the ideas depends on the quality of the thinking that went into them, of course, but you won't know what kinds of thinkers are out there until you stimulate them with a challenging problem.

Naturally, there are some discussions that must be had privately; throughout this book we'll see examples of those. But the guiding principle should always be: *If there's no reason for it to be private, it should be public.*

Making this happen requires action. It's not enough merely to ensure that all your own posts go to the public list. You also have to nudge other people's unnecessarily private conversations to the list too. If someone tries to start a private discussion with you and there's no reason for it to be private, then it is incumbent on you to open the appropriate meta-discussion immediately. Don't even comment on the original topic until you've either successfully steered the conversation to a public place, or ascertained that privacy really was needed. If you do this consistently, people will catch on pretty quickly and start to use the public forums by default.

Nip Rudeness in the Bud

From the very start of your project's public existence, you should maintain a zero-tolerance policy toward rude or insulting behavior in its forums. Zero-tolerance does not mean technical enforcement per se. You don't have to remove people from the mailing list when they flame another subscriber, or take away their commit access because they made derogatory comments. (In theory, you might eventually have to resort to such actions, but only after all other avenues have failed—which, by definition, isn't the case at the start of the project.) Zero-tolerance simply means never letting bad behavior slide by unnoticed. For example, when someone posts a technical comment mixed together with an *ad hominem* at-

[10] We haven't gotten to the section on crediting yet, but just to practice what I'll later preach: the observer's name was Brian Behlendorf, and he was emphatic about the general importance of keeping all discussions public unless there was a specific need for privacy.

tack on some other developer in the project, it is imperative that your response address the *ad hominem* attack as a separate issue unto itself, separate from the technical content.

It is unfortunately very easy, and all too typical, for constructive discussions to lapse into destructive flame wars. People will say things in email that they would never say face-to-face. The topics of discussion only amplify this effect: in technical issues, people often feel there is a single right answer to most questions, and that disagreement with that answer can only be explained by ignorance or stupidity. It's a short distance from calling someone's technical proposal stupid to calling the person themselves stupid. In fact, it's often hard to tell where technical debate leaves off and character attack begins, which is one reason why drastic responses or punishments are not a good idea. Instead, when you think you see it happening, make a post that stresses the importance of keeping the discussion friendly, without accusing anyone of being deliberately poisonous. Such "Nice Police" posts do have an unfortunate tendency to sound like a kindergarten teacher lecturing a class on good behavior:

> *First, let's please cut down on the (potentially) ad hominem comments; for example, calling J's design for the security layer "naive and ignorant of the basic principles of computer security." That may be true or it may not, but in either case it's no way to have the discussion. J made his proposal in good faith. If it has deficiencies, point them out, and we'll fix them or get a new design. I'm sure M meant no personal insult to J, but the phrasing was unfortunate, and we try to keep things constructive around here.*
>
> *Now, on to the proposal. I think M was right in saying that...*

As stilted as such responses sound, they have a noticeable effect. If you consistently call out bad behavior, but don't demand an apology or acknowledgment from the offending party, then you leave people free to cool down and show their better side by behaving more decorously next time—and they will.

One of the secrets of doing this successfully is to never make the meta-discussion the main topic. It should always be an aside, a brief preface to the main portion of your response. Point out in passing that "we don't do things that way around here," but then move on to the real content, so that you're giving people something on-topic to respond to. If someone protests that they didn't deserve your rebuke, simply refuse to be drawn into an argument about it. Either don't respond (if you think they're just letting off steam and don't require a response), or say you're sorry if you overreacted and that it's hard to detect nuance in email, then get back to the main topic. Never, ever insist on an acknowledgment, whether public or private, from someone that they behaved inappropriately. If they choose of their own volition to post an apology, that's great, but demanding that they do so will only cause resentment.

The overall goal is to make good etiquette be seen as one of the "in-group" behaviors. This helps the project, because developers can be driven away (even from projects they like and want to support) by flame wars. You may not even know that they were driven away; someone might lurk on the mailing list, see that it takes a thick skin to participate in the project, and decide against getting involved at all. Keeping forums friendly is a long-term survival strategy, and it's easier to do when the project is still small. Once it's part of the culture, you won't have to be the only person promoting it. It will be maintained by everyone.

Codes of Conduct

In the decade since the first edition of this book in 2006, it has become somewhat more common for open source projects, especially the larger ones, to adopt an explicit *code of conduct*. I think this is a good trend. As open source projects become, at long last, more diverse, the presence of a code of conduct can remind participants to think twice about whether a joke is going to be hurtful to some people, or whether — to pick a random example — it contributes to a welcoming and inclusive atmosphere when an open source image processing library's documentation just happens to use yet another picture of a

pretty young woman to illustrate the behavior of a particular algorithm. Codes of conduct remind participants that the maintenance of a respectful and welcoming environment is everyone's responsibility.

An Internet search will easily find many examples of codes of conduct for open source projects. The most popular one is probably the one at contributor-covenant.org [https://contributor-covenant.org/], so naturally there's a positive feedback dynamic if you choose that one: more developers will be already familiar with it, plus you get its translations into other languages for free, etc.

A code of conduct will *not* solve all the interpersonal problems in your project. Furthermore, if it is misused, it has the potential to create new problems — it's always possible to find people who specialize in manipulating social norms and rules to harm a community rather than help it (see the section called “Difficult People” in Chapter 6, *Communications*), and if you're particularly unlucky some of those people may find their way into your project. It is always up to the project leadership, by which I mean those whom others in the project tend to listen to the most, to see to it that a code of conduct is used wisely. (See also the section called “Recognizing Rudeness” in Chapter 6, *Communications*.)

Some participants may genuinely disagree with the need to adopt a code at all, and argue against it on the grounds that it could do more harm than good. Even if you feel they're wrong, it is imperative that you help make sure they're able to state their view without being attacked for it. After all, disagreeing with the need for a code of conduct is not the same as — is, in fact, entirely unrelated to — engaging in behavior that would be a violation of the proposed code of conduct. Sometimes people confuse these two things, and need to be reminded of the distinction.[11]

Practice Conspicuous Code Review

One of the best ways to foster a productive development community is to get people looking at each others' code — ideally, to get them looking at each others' code *changes* as those changes arrive. *Commit review* (sometimes just called *code review*) is the practice of reviewing commits as they come in, looking for bugs and possible improvements.

There are a couple of reasons to focus on reviewing changes, rather than on reviewing code that's been around for a while. First, it just works better socially: when someone reviews your change, she is interacting with work you did recently. That means if she comments on it right away, you will be maximally interested in hearing what she has to say; six months later, you might not feel as motivated to engage, and in any case might not remember the change very well. Second, looking at what changes in a codebase is a gateway to looking at the rest of the code anyway — reviewing a change often causes one to look at the surrounding code, at the affected callers and callees elsewhere, at related module interfaces, etc.[12]

Commit review thus serves several purposes simultaneously. It's the most obvious example of peer review in the open source world, and directly helps to maintain software quality. Every bug that ships in a piece of software got there by being committed and not detected; therefore, the more eyes watch commits, the fewer bugs will ship. But commit review also serves an indirect purpose: it confirms to people that what they do matters, because one obviously wouldn't take time to review a commit unless one cared about its effect. People do their best work when they know that others will take the time to evaluate it.

Reviews should be public. Even on occasions when I have been sitting in the same physical room with another developer, and one of us has made a commit, we take care not to do the review verbally in the

[11] There's an excellent post entitled "The complex reality of adopting a meaningful code of conduct" [https://subfictional.com/2016/01/25/the-complex-reality-of-adopting-a-meaningful-code-of-conduct/] by Christie Koehler, discussing this in much more depth.

[12] None of this is an argument against top-to-bottom code review, of course, for example to do a security audit. But while that kind of review is important too, it's more of a generic development best practice, and is not as specifically relevant to running an open source project as change-by-change review is.

room, but to send it to the appropriate online review forum instead. Everyone benefits from seeing the review happen. People follow the commentary and sometimes find flaws in it; even when they don't, it still reminds them that review is an expected, regular activity, like washing the dishes or mowing the lawn.

Some technical infrastructure is required to do change-by-change review effectively. In particular, setting up commit notifications is extremely useful. The effect of commit notifications is that every time someone commits a change to the central repository, an email or other subscribable notification goes out showing the log message and diffs (unless the diff is too large; see *diff*, in the section called “Version Control Vocabulary”). The review itself might take place on a mailing list, or in a review tool such as Gerrit or the GitHub "pull request" interface. See the section called “Commit notifications / commit emails” in Chapter 3, *Technical Infrastructure* for details.

Case study

In the Subversion project, we did not at first make a regular practice of code review. There was no guarantee that every commit would be reviewed, though one might sometimes look over a change if one were particularly interested in that area of the code. Bugs slipped in that really could and should have been caught. A developer named Greg Stein, who knew the value of code review from past work, decided that he was going to set an example by reviewing every line of *every single commit* that went into the code repository. Each commit anyone made was soon followed by an email to the developer's list from Greg, dissecting the commit, analyzing possible problems, and occasionally praising a clever bit of code. Right away, he was catching bugs and non-optimal coding practices that would otherwise have slipped by without ever being noticed. Pointedly, he never complained about being the only person reviewing every commit, even though it took a fair amount of his time, but he did sing the praises of code review whenever he had the chance. Pretty soon, other people, myself included, started reviewing commits regularly too.

What was our motivation? It wasn't that Greg had consciously shamed us into it. But he had proven that reviewing code was a valuable way to spend time, and that one could contribute as much to the project by reviewing others' changes as by writing new code. Once he demonstrated that, it became expected behavior, to the point where any commit that didn't get some reaction would cause the committer to worry, and even ask on the list whether anyone had had a chance to review it yet. Later, Greg got a job that didn't leave him as much time for Subversion, and had to stop doing regular reviews. But by then, the habit was so ingrained for the rest of us as to seem that it had been going on since time immemorial.

Start doing reviews from very first commit. The sorts of problems that are easiest to catch by reviewing diffs are security vulnerabilities, memory leaks, insufficient comments or API documentation, off-by-one errors, caller/callee discipline mismatches, and other problems that require a minimum of surrounding context to spot. However, even larger-scale issues such as failure to abstract repeated patterns to a single location become spottable after one has been doing reviews regularly, because the memory of past diffs informs the review of present diffs.

Don't worry that you might not find anything to comment on, or that you don't know enough about every area of the code. There will usually be something to say about almost every commit; even where you don't find anything to question, you may find something to praise. The important thing is to make it clear to every committer that what they do is seen and understood, that attention is being paid. Of course, code review does not absolve programmers of the responsibility to review and test their changes before committing; no one should depend on code review to catch things she ought to have caught on her own.

Be Open From Day One

Start your project out in the open from the very first day. The longer a project is run in a closed source manner, the harder it is to open source later.[13]

Being open source from the start doesn't mean your developers must immediately take on the extra responsibilities of community management. People often think that "open source" means "strangers distracting us with questions", but that's optional — it's something you might do down the road, if and when it makes sense for your project. It's under your control. There are still major advantages to be had by running the project out in open, publicly-visible forums from the beginning. Conversely, the longer the project is run closed-source, the more difficult it will be to open up later.

I think there's one underlying cause for this:

At each step in a project, programmers face a choice: to do that step in a manner compatible with a hypothetical future open-sourcing, or do it in a manner incompatible with open-sourcing. And every time they choose the latter, the project gets just a little bit harder to open source.

The crucial thing is, they can't help choosing the latter occasionally — all the pressures of development propel them that way. It's very difficult to give a future event the same present-day weight as, say, fixing the incoming bugs reported by the testers, or finishing that feature the customer just added to the spec. Also, programmers struggling to stay on budget will inevitably cut corners here and there (in Ward Cunningham's phrase, they will incur "technical debt [https://en.wikipedia.org/wiki/Technical_debt]"), with the intention of cleaning it up later.

Thus, when it's time to open source, you'll suddenly find there are things like:

- Customer-specific configurations and passwords checked into the code repository;
- Sample data constructed from live (and confidential) information;
- Bug reports containing sensitive information that cannot be made public;
- Comments in the code expressing perhaps overly-honest reactions to the customer's latest urgent request;
- Archives of correspondence among the developer team, in which useful technical information is interleaved with personal opinions not intended for strangers;
- Licensing issues due to dependency libraries whose terms might have been fine for internal deployment (or not even that), but aren't compatible with open source distribution;
- Documentation written in the wrong format (e.g., that proprietary internal wiki your department uses), with no easy translation tool available to get it into formats appropriate for public distribution;
- Non-portable build dependencies that only become apparent when you try to move the software out of your internal build environment;
- Modularity violations that everyone knows need cleaning up, but that there just hasn't been time to take care of yet...
- (This list could go on.)

The problem isn't just the work of doing the cleanups; it's the extra decision-making they sometimes require. For example, if sensitive material was checked into the code repository in the past, your team now

[13]This section started out as a blog post, "Be Open from Day One, not Day N." [http://archive.civiccommons.org/2011/01/be-open-from-day-one/index.html], though it's been edited a lot for inclusion here.

faces a choice between cleaning it out of the historical revisions entirely, so you can open source the entire (sanitized) history, or just cleaning up the latest revision and open-sourcing from that (sometimes called a "top-skim"). Neither method is wrong or right — and that's the problem: now you've got one more discussion to have and one more decision to make. In some projects, that decision gets made and reversed several times before the final release. The thrashing itself is part of the cost.

Waiting Just Creates an Exposure Event

The other problem with opening up a developed codebase is that it creates a needlessly large exposure event. Whatever issues there may be in the code (modularity corner-cutting, security vulnerabilities, etc), they are all exposed to public scrutiny at once — the open-sourcing event becomes an opportunity for the technical blogosphere to pounce on the code and see what they can find.

Contrast that with the scenario where development was done in the open from the beginning: code changes come in one at a time, so problems are handled as they come up (and are often caught sooner, since there are more eyeballs on the code). Because changes reach the public at a low, continuous rate of exposure, no one blames your development team for the occasional corner-cutting or flawed code checkin. Everyone's been there, after all; these tradeoffs are inevitable in real-world development. As long as the technical debt is properly recorded in "FIXME" comments and bug reports, and any security issues are addressed promptly, it's fine. Yet if those same issues were to appear suddenly all at once, unsympathetic observers may jump on the aggregate exposure in a way they never would have if the issues had come up piecemeal in the normal course of development.

(These concerns apply even more strongly to government software projects; see the section called “Being Open Source From Day One is Especially Important for Government Projects” in the section called “Governments and Open Source”.)

The good news is that these are all unforced errors. A project incurs little extra cost by avoiding them in the simplest way possible: by running in the open from Day One.

"In the open" means the following things are publicly accessible, in standard formats, from the first day of the project: the code repository, bug tracker, design documents, user documentation, wiki, and developer discussion forums. It also means the code and documentation are placed under an open source license, of course. It also means your team's day-to-day work takes place in the publicly visible area.

"In the open" does not have to mean: allowing strangers to check code into your repository (they're free to copy it into their own repository, if they want, and work with it there); allowing anyone to file bug reports in your tracker (you're free to choose your own QA process, and if allowing reports from strangers doesn't help you, you don't have to do it); reading and responding to every bug report filed, even if you do allow strangers to file; responding to every question people ask in the forums (even if you moderate them through); reviewing every patch or suggestion posted, when doing so may cost valuable development time; etc.

One way to think of it is that you're open sourcing your code, not your time. One of those resources is infinitely replicable, the other is not. You'll have to determine the point at which engaging with outside users and developers makes sense for your project. In the long run it usually does, and most of this book is about how to do it effectively. But it's still under your control. Developing in the open does not change this, it just ensures that everything done in the project is, by definition, done in a way that's compatible with being open source.

Opening a Formerly Closed Project

As per the section called “Be Open From Day One”, it's best to avoid being in the situation of opening up a closed project in the first place; just start the project in the open if you can. But if it's too late for

that, and you find yourself opening up an existing project that already has active developers accustomed to working in a closed-source environment, there are certain common issues that tend to arise. You can save a lot of time and trouble if you are prepared for them.

Some of these issues are essentially mechanical — indeed, be-open-from-day-one can serve as a checklist of sorts. For example, if your code depends on proprietary libraries that are not part of the standard distribution of your target operating system(s), you will need to find open source replacements; if there is confidential content — e.g., unpublishable comments, passwords or site-specific configuration information that cannot easily be changed, confidential data belonging to third parties, etc — in the project's version control history, then you may have to release a "top-skim" version, that is, restart the version history afresh from the current version as of the moment you open source the code; and so on.

But there can be social and managerial issues too, and they are often more significant in the long run than the mere mechanical concerns. You will need to make sure everyone on the development team understands that a big change is coming—and make sure that you understand how it's going to feel from their point of view.

Try to imagine how the situation looks to them: formerly, all code and design decisions were made with a group of other programmers who knew the software more or less equally well, who all received the same pressures from the same management, and who all know each others' strengths and weaknesses. Now you're asking them to expose their code to the scrutiny of random strangers, who will form judgements based only on the code, with no awareness of what business pressures may have forced certain decisions. These strangers will ask lots of questions, questions that jolt the existing developers into realizing that the documentation they slaved so hard over is *still* inadequate (this is inevitable). To top it all off, the newcomers are unknown, faceless entities. If one of your developers already feels insecure about his skills, imagine how that will be exacerbated when newcomers point out flaws in code he wrote, and worse, do so in front of his colleagues. Unless you have a team of perfect coders, this is unavoidable—in fact, it will probably happen to all of them at first. This is not because they're bad programmers; it's just that any program above a certain size has bugs, and peer review will spot some of those bugs (see the section called “Practice Conspicuous Code Review” earlier in this chapter). At the same time, the newcomers themselves won't be subject to much peer review at first, since they can't contribute code until they're more familiar with the project. To your developers, it may feel like all the criticism is incoming, never outgoing. Thus, there is the danger of a siege mentality taking hold among the old hands.

The best way to prevent this is to warn everyone about what's coming, explain it, tell them that the initial discomfort is perfectly normal, and reassure them that it's going to get better. Some of these warnings should take place privately, before the project is opened. But you may also find it helpful to remind people on the public lists that this is a new way of development for the project, and that it will take some time to adjust. The very best thing you can do is lead by example. If you don't see your developers answering enough newbie questions, then just telling them to answer more isn't going to help. They may not have a good sense of what warrants a response and what doesn't yet, or it could be that they don't have a feel for how to prioritize coding work against the new burden of external communications. The way to get them to participate is to participate yourself. Be on the public mailing lists, and make sure to answer some questions there. When you don't have the expertise to field a question, then visibly hand it off to a developer who does—and watch to make sure he follows up with an answer, or at least a response. It will naturally be tempting for the longtime developers to lapse into private discussions, since that's what they're used to. Make sure you're subscribed to the internal mailing lists on which this might happen, so you can ask that such discussions be moved to the public lists right away.

If you expect the newly-public project to start involving developers who are not paid directly for their work — and there are usually at least a few such developers on most successful open source projects — see Chapter 5, *Organizations, Money, and Business* for discussion of how to mix paid and unpaid developers successfully.

Announcing

Once the project is presentable—not perfect, just presentable—you're ready to announce it to the world.

This is a simpler process than you might expect. First, set up the announcement pages at your project's home site, as described in the section called “Announcing Releases and Other Major Events”) in Chapter 6, *Communications*. Then, post announcements in the appropriate forums. There are two kinds of forums: generic forums that display many kinds of new project announcements, and topic-specific forums where your project would be welcome news.

As of late 2015, the best general forum is probably news.ycombinator.com [https://news.ycombinator.com/]. While you are welcome to submit your project there, note that it will have to successfully climb the word-of-mouth / upvote tree to get featured on the front page. The subreddit forums related to reddit.com/r/opensource [https://www.reddit.com/r/opensource/], reddit.com/r/programming [https://www.reddit.com/r/programming/], and reddit.com/r/software [https://www.reddit.com/r/software/] work in a similar way. While it's good news for your project if you can get mentioned in a place like that, I hesitate to contribute to the marketing arms race by suggesting any concrete steps to accomplish this. Use your judgement and try not to spam.

Send the announcement to changelog.com [https://changelog.com/]; if they think it's interesting to enough people, they'll give it some coverage, and that can get you some good initial developer attention. You might also try registering your project at openhatch.org [https://openhatch.org/], a site that matches open source projects with programmers looking for good projects to participate in, and at openhub.net [http://openhub.net/], which is the closest thing there is to an global database of free software projects and their contributors, though it is somewhat desultorily maintained.

Finally, topic-specific forums are probably where you'll get the most interest. Think of mailing lists or web forums where an announcement of your project would be on-topic and of interest — you might already be a member of some of them — and post there. Be careful to make exactly *one* post per forum, and to direct people to your project's own discussion areas for follow-up discussion (when posting by email, you can do this by setting the `Reply-to` header). Your announcement should be short and get right to the point, and the Subject line should make it clear that it is an announcement of a new project:

```
To: discuss@some.forum.about.search.indexers
Subject: [ANN] Scanley, a new full-text indexer project.
Reply-to: dev@scanley.org

This is a one-time post to announce the creation of the Scanley
project, an open source full-text indexer and search engine with a
rich API, for use by programmers in providing search services for
large collections of text files.  Scanley is now running code, is
under active development, and is looking for both developers and
testers.

Home page: http://www.scanley.org/

Features:
   - Searches plain text, HTML, and XML
   - Word or phrase searching
   - (planned) Fuzzy matching
   - (planned) Incremental updating of indexes
   - (planned) Indexing of remote web sites
   - (planned) Long-distance mind-reading
```

```
Requirements:
   - Python 3.2 or higher
   - SQLite 3.8.1 or higher

For more information, please come find us at scanley.org!

Thank you,
-J. Random
```

(See the section called “Publicity” in Chapter 6, *Communications* for advice on announcing subsequent releases and other project events.)

There is an ongoing debate in the free software world about whether it is necessary to begin with running code, or whether a project can benefit from being announced even during the design/discussion stage. I used to think starting with running code was crucial, that it was what separated successful projects from toys, and that serious developers would only be attracted to software that already does something concrete.

This turned out not to be the case. In the Subversion project, we started with a design document, a core of interested and well-connected developers, a lot of fanfare, and *no* running code at all. To my complete surprise, the project acquired active participants right from the beginning, and by the time we did have something running, there were quite a few developers already deeply involved. Subversion is not the only example; the Mozilla project was also launched without running code, and is now a successful and popular web browser.

On the evidence of this and other examples, I have to back away from the assertion that running code is absolutely necessary for launching a project. Running code is still the best foundation for success, and a good rule of thumb would be to wait until you have it before announcing your project[14]. However, there may be circumstances where announcing earlier makes sense. I do think that at least a well-developed design document, or else some sort of code framework, is necessary—of course it may be revised based on public feedback, but there has to be something concrete, something more tangible than just good intentions, for people to sink their teeth into.

Whenever you announce, don't expect a horde of participants to join the project immediately afterward. Usually, the result of announcing is that you get a few casual inquiries, a few more people join your mailing lists, and aside from that, everything continues pretty much as before. But over time, you will notice a gradual increase in participation from both new code contributors and users. Announcement is merely the planting of a seed. It can take a long time for the news to spread. If the project consistently rewards those who get involved, the news *will* spread, though, because people want to share when they've found something good. If all goes well, the dynamics of exponential communications networks will slowly transform the project into a complex community, where you don't necessarily know everyone's name and can no longer follow every single conversation. The next chapters are about working in that environment.

[14]Note that *announcing* your project can come long after you have open sourced the code. My advice to consider carefully the timing of your announcement should not be taken as advice to delay open sourcing the code — ideally, your project should be open source and publicly visible from the very first moment of its existence, and this is entirely independent of when you announce it. See the section called “Be Open From Day One” for more.

Chapter 3. Technical Infrastructure

Free software projects rely on technologies that support the selective capture and integration of information. The more skilled you are at using these technologies, and at persuading others to use them, the more successful your project will be.

This only becomes more true as the project grows. Good information management is what prevents open source projects from collapsing under the weight of Brooks' Law[1], which states that adding manpower to a late software project makes it later. Fred Brooks observed that the complexity of communications in a project increases as the *square* of the number of participants. When only a few people are involved, everyone can easily talk to everyone else, but when hundreds of people are involved, it is no longer possible for each person to remain constantly aware of what everyone else is doing. If good free software project management is about making everyone feel like they're all working together in the same room, the obvious question is: what happens when everyone in a crowded room tries to talk at once?

This problem is not new. In non-metaphorical crowded rooms, the solution is *parliamentary procedure*: formal guidelines for how to have real-time discussions in large groups, how to make sure important dissents are not lost in floods of "me-too" comments, how to form subcommittees, how to recognize and record when decisions are made, etc. An important part of parliamentary procedure is specifying how the group interacts with its information management system. Some remarks are made "for the record", others are not. The record itself is subject to direct manipulation, and is understood to be not a literal transcript of what occurred, but a representation of what the group is willing to *agree* occurred. The record is not monolithic, but takes different forms for different purposes. It comprises the minutes of individual meetings, the complete collection of all minutes of all meetings, summaries, agendas and their annotations, committee reports, reports from correspondents not present, lists of action items, etc.

Because the Internet is not really a room, we don't have to worry about replicating those parts of parliamentary procedure that keep some people quiet while others are speaking. But when it comes to information management techniques, well-run open source projects are parliamentary procedure on steroids. Since almost all communication in open source projects happens in writing, elaborate systems have evolved for routing and labeling data appropriately, for minimizing repetitions so as to avoid spurious divergences, for storing and retrieving data, for correcting bad or obsolete information, and for associating disparate bits of information with each other as new connections are observed. Active participants in open source projects internalize many of these techniques, and will often perform complex manual tasks to ensure that information is routed correctly. But the whole endeavor ultimately depends on sophisticated software support. As much as possible, the communications media themselves should do the routing, labeling, and recording, and should make the information available to humans in the most convenient way possible. In practice, of course, humans will still need to intervene at many points in the process, and it's important that the software make such interventions convenient too. But in general, if the humans take care to label and route information accurately on its first entry into the system, then the software should be configured to make as much use of that metadata as possible.

The advice in this chapter is intensely practical, based on experiences with specific software and usage patterns. But the point is not just to teach a particular collection of techniques. It is also to demonstrate, by means of many small examples, the overall attitude that will best encourage good information management in your project. This attitude will involve a combination of technical skills and people skills. The technical skills are essential because information management software always requires configuration, plus a certain amount of ongoing maintenance and tweaking as new needs arise (for example, see the discussion of how to handle project growth in the section called “Pre-Filtering the Bug Tracker” later in this chapter). The people skills are necessary because the human community also requires mainte-

[1]From his book *The Mythical Man Month*, 1975. See en.wikipedia.org/wiki/The_Mythical_Man-Month [https://en.wikipedia.org/wiki/The_Mythical_Man-Month], en.wikipedia.org/wiki/Brooks_Law [https://en.wikipedia.org/wiki/Brooks_Law], and en.wikipedia.org/wiki/Fred_Brooks [https://en.wikipedia.org/wiki/Fred_Brooks].

nance: it's not always immediately obvious how to use these tools to full advantage, and in some cases projects have conflicting conventions (for example, see the discussion of setting `Reply-to` headers on outgoing mailing list posts, in the section called “Mailing Lists / Message Forums”). Everyone involved with the project will need to be encouraged, at the right times and in the right ways, to do their part to keep the project's information well organized. The more involved the contributor, the more complex and specialized the techniques she can be expected to learn.

Information management has no cut-and-dried solution. There are too many variables. You may finally get everything configured just the way you want it, and have most of the community participating, but then project growth will make some of those practices unscalable. Or project growth may stabilize, and the developer and user communities settle into a comfortable relationship with the technical infrastructure, but then someone will come along and invent a whole new information management service, and pretty soon newcomers will be asking why your project doesn't use it—for example, this happened to a lot of free software projects that predate the invention of the wiki (see en.wikipedia.org/wiki/Wiki [https://en.wikipedia.org/wiki/Wiki]), and more recently has been happening to projects whose workflows were developed before the rise of GitHub PRs (see the section called “Pull requests”) as the canonical way to package proposed contributions. Many infrastructure questions are matters of judgement, involving tradeoffs between the convenience of those producing information and the convenience of those consuming it, or between the time required to configure information management software and the benefit it brings to the project.

Beware of the temptation to over-automate, that is, to automate things that really require human attention. Technical infrastructure is important, but what makes a free software project work is care—and intelligent expression of that care—by the humans involved. The technical infrastructure is mainly about giving humans easy ways to apply care.

What a Project Needs

Most open source projects offer at least a minimum, standard set of tools for managing information:

Web site

: Primarily a centralized, one-way conduit of information from the project out to the public. The web site may also serve as an administrative interface for other project tools.

Mailing lists / Message forums

: Usually the most active communications forum in the project, and the "medium of record."

Version control

: Enables developers to manage code changes conveniently, including reverting and "change porting". Enables everyone to watch what's happening to the code.

Bug tracking

: Enables developers to keep track of what they're working on, coordinate with each other, and plan releases. Enables everyone to query the status of bugs and record information (e.g., reproduction recipes) about particular bugs. Can be used for tracking not only bugs, but also tasks, releases, new features, etc.

Real-time chat

: A place for quick, lightweight discussions and question/answer exchanges. Not always archived completely.

Each tool in this set addresses a distinct need, but their functions are also interrelated, and the tools must be made to work together. Below we will examine how they can do so, and more importantly, how to get people to use them.

You may be able to avoid a lot of the headache of choosing and configuring many of these tools by using a *canned hosting* site: an online service that offers prepackaged, templatized web services with some or all of the collaboration tools needed to run a free software project. See the section called “Canned Hosting” later in this chapter for a discussion of the advantages and disadvantages of canned hosting.

Web Site

For our purposes, *the web site* means web pages devoted to helping people participate in the project as developers, documenters, etc. Note that this is different from the main user-facing web site. In many projects, users have different needs and often (statistically speaking) a different mentality from the developers. The kinds of web pages most helpful to users are not necessarily the same as those helpful for developers — don't try to make a "one size fits all" web site just to save some writing and maintenance effort: you'll end up with a site that is not quite right for either audience.

The two types of sites should cross-link, of course, and in particular it's important that the user-oriented site have, tucked a way in a corner somewhere, a clear link to the developers' site, since most new developers will start out at the user-facing pages and look for a path from there to the developers' area.

An example may make this clearer. As of this writing in November 2013, the office suite LibreOffice has its main user-oriented web site at libreoffice.org [https://libreoffice.org/], as you'd expect. If you were a user wanting to download and install LibreOffice, you'd start there, go straight to the "Download" link, and so on. But if you were a developer looking to (say) fix a bug in LibreOffice, you might *start* at libreoffice.org [https://libreoffice.org/], but you'd be looking for a link that says something like "Developers", or "Development", or "Get Involved" — in other words, you'd be looking for the gateway to the development area.

In the case of LibreOffice, as with some other large projects, they actually have a couple of different gateways. There's one link that says "Get Involved" [https://www.libreoffice.org/get-involved/], and another that says "Developers" [https://www.libreoffice.org/developers/]. The "Get Involved" page is aimed at the broadest possible range of potential contributors: developers, yes, but also documenters, quality-assurance testers, marketing helpers, web infrastructure experts, financial or in-kind donors, interface designers, support forum helpers, etc. This frees up the "Developers" page to target the rather narrower audience of programmers who want to get involved in improving the LibreOffice code. The set of links and short descriptions provided on both pages is admirably clear and concise: you can tell immediately from looking whether you're in the right place for what you want do, and if so what the next thing to click on is. The "Development" page gives some information about where to find the code, how to contact the other developers, how to file bugs, and things like that, but most importantly it points to what most seasoned open source contributors would instantly recognize as the *real* gateway to actively-maintained development information: the development wiki at wiki.documentfoundation.org/Development [https://wiki.documentfoundation.org/Development].

This division into two contributor-facing gateways, one for all kinds of contributions and another for coders specifically, is probably right for a large, multi-faceted project like LibreOffice. You'll have to use your judgement as to whether that kind of subdivision is appropriate for your project; at least at the beginning, it probably isn't. It's better to start with one unified contributor gateway, aimed at all the types of contributors you expect, and if that page ever gets large enough or complex enough to feel unwieldy — listen carefully for complaints about it, since you and other long-time participants will be naturally desensitized to weaknesses in introductory pages! — then you can divide it up however seems best.

From a technical point of view there is not much to say about setting up the project web site. Configuring a web server and writing web pages are fairly simple tasks, and most of the important things to say about layout and arrangement were covered in the previous chapter. The web site's main function is to present a clear and welcoming overview of the project, and to bind together the other tools (the version control system, bug tracker, etc.). If you don't have the expertise to set up a web server yourself, it's usually not hard to find someone who does and is willing to help out. Nonetheless, to save time and effort, people often prefer to use one of the canned hosting sites.

Canned Hosting

A *canned hosting* site is an online service that offers some or all of the online collaboration tools needed to run a free software project. At a minimal, a canned hosting site offers public version control repositories and bug tracking; most also offer wiki space, many offer mailing list hosting too, and some offer continuous integration testing and other services.[2]

There are two main advantages to using a canned site. The first is server capacity and bandwidth: their servers are hefty boxes sitting on really fat pipes. No matter how successful your project gets, you're not going to run out of disk space or swamp the network connection. The second advantage is simplicity. They have already chosen a bug tracker, a version control system, perhaps discussion forum software, and everything else you need to run a project. They've configured the tools, arranged single-sign-on authentication where appropriate, are taking care of backups for all the data stored in the tools, etc. You don't need to make many decisions. All you have to do is fill in a registration form, press a button, and suddenly you've got a project development web site.

These are pretty significant benefits. The disadvantage, of course, is that you must accept *their* choices and configurations, even if something different would be better for your project. Usually canned sites are adjustable within certain narrow parameters, but you will never get the fine-grained control you would have if you set up the site yourself and had full administrative access to the server.

A perfect example of this is the handling of generated files. Certain project web pages may be generated files—for example, there are systems for keeping FAQ data in an easy-to-edit master format, from which HTML, PDF, and other presentation formats can be generated. As explained in the section called “Version everything” earlier in this chapter, you wouldn't want to version the generated formats, only the master file. But when your web site is hosted on someone else's server, it may be difficult to set up a custom hook to regenerate the online HTML version of the FAQ whenever the master file is changed.

If you choose a canned site, try to leave open the option of switching to a different site later, by using a custom domain name as the project's development home address. You can forward that URL to the canned site, or have a fully customized development home page at the main URL and link to the canned site for specific functionality. Just try to arrange things such that if you later decide to use a different hosting solution, the project's main address doesn't need to change.

And if you're not sure whether to use canned hosting, then you should probably use canned hosting. These sites have integrated their services in myriad ways (just one example: if a commit mentions a bug ticket number using a certain format, then people browsing that commit later will find that it automatically links to that ticket), ways that would be laborious for you to reproduce, especially if it's your first time running an open source project. The universe of possible configurations of collaboration tools is vast and complex, but the same set of choices has faced everyone running an open source project and there are some settled solutions now. Each of the canned hosting sites implements a reasonable subset of that solution space, and unless you have reason to believe you can do better, your project will probably run best just using one of those sites.

[2]Note that for successful free software projects, interested commercial entities will eventually often step to fund many of these services anyway; see the section called “Providing Build Farms and Development Servers” for further discussion of this.

Choosing a canned hosting site

There are now so many sites providing free-of-charge canned hosting for projects released under open source licenses that there is not space here to review the field.

So I'll make this easy:

If you don't know what to choose, then choose GitHub [https://github.com/]. It's by far the most popular and appears set to stay that way for some years to come. It has a good set of features and integrations. Many developers are already familiar with GitHub and have an account there. It has an API [https://develop.github.com/] for interacting programmatically with project resources, and while it does not currently offer mailing lists, there are plenty of other places you can provision those, such as discourse.org [https://discourse.org/] and Google Groups [https://groups.google.com/].

If you're not convinced by GitHub (for example because your project uses, say, Mercurial instead of Git for version control), but you aren't sure where to host, take a look at Wikipedia's thorough comparison of open source software hosting facilities [https://en.wikipedia.org/wiki/Comparison_of_open_source_software_hosting_facilities]; it's the first place to look for up-to-date, comprehensive information on open source project hosting options.

Hosting on fully open source infrastructure

Although all the canned hosting sites use plenty of free software in their stack, most of them also wrote some proprietary code to glue it all together. In these cases the hosting environment itself is not fully open source, and thus cannot be easily reproduced by others. For example, while Git itself is free software, GitHub is a hosted service running partly with proprietary software — if you leave GitHub, you can't take a copy of their infrastructure with you, at least not all of it.

Some projects would prefer a canned hosting site that runs an entirely free software infrastructure and that could, in theory, be reproduced independently were that ever to become necessary. Fortunately, there are a few such sites. As of mid-2015, phacility.com [https://phacility.com] offers low-cost project hosting in Phabricator [http://phabricator.org/], a fully open source web platform for open-source style collaboration.

There is also GitLab [https://gitlab.com/], which until recently was fully open source. However, they now seem to offer an open source edition for self-hosting and an "enterprise edition" that they host and that has a few features the open source edition doesn't have. Unfortunately, GitLab.com doesn't currently seem to offer a hosting package based on the strictly open source edition, which is too bad, because that would be the best choice for an open source project that wants to outsource hosting while preserving its commitment, both philosophical and utilitarian, to software freedom.

Furthermore, any service that offers hosting of the Redmine [http://www.redmine.org/] or Trac [https://trac.edgewall.org/] code collaboration platforms effectively offers fully freedom-preserving project hosting, because those platforms include most of the features needed to run an open source project. Some companies offer that kind of commercial platform hosting with a zero-cost or very cheap rate for open source projects.

Should you host your project on fully open source infrastructure? While it would be ideal, from the free software philosophical perspective, to have access to all the code that runs the site, I cannot say I've seen any evidence that hosting or not hosting on that kind of site has any effect on a project's success. The vast majority of developers who work on free software projects are willing to participate through a non-free hosting platform when that's what the project is using.

In my opinion, the crucial thing is to be able to interact with project data in automatable ways, and to have a way to export the data. A site that meets these criteria can never truly lock you in, and will even

be somewhat extensible, via its programmatic interface. While there is some value in having all the code that runs a hosting site available under open source terms, in practice the demands of actually deploying that code in a production environment are difficult and sometimes prohibitive for most projects anyway. The largest of these sites need multiple servers, customized networks, and full-time staffs to keep them running; merely having the code would not be sufficient to duplicate or "fork" them anyway. The main thing is just to make sure your data isn't trapped.

Of course, all the above applies only to the servers of the hosting site. Your project itself should never require participants to run proprietary collaboration software on their own machines.[3]

Anonymity and involvement

A problem that is not strictly limited to the canned sites, but is most often found there, is the over-requirement of user registration to participate in various aspects of the project. The proper degree of requirement is a bit of a judgement call. User registration helps prevent spam, for one thing, and even if every commit gets reviewed you still probably don't want anonymous strangers pushing changes into your repository, for example.

But sometimes user registration ends up being required for tasks that ought to be permitted to unregistered visitors, especially the ability to file tickets in the bug tracker, and to comment on existing tickets. By requiring a logged-in username for such actions, the project raises the involvement bar for what should be quick, convenient tasks. It also changes the demographics of who files bugs, since those who take the trouble to set up a user account at the project site are hardly a random sample even from among users who are willing to file bugs (who in turn are already a biased subset of all the project's users). Of course, one wants to be able to contact someone who's entered data into the ticket tracker, but having a field where she can enter her email address (if she wants to) is sufficient. If a new user spots a bug and wants to report it, she'll only be annoyed at having to fill out an account creation form before she can enter the bug into the tracker. She may simply decide not to file the bug at all.

If you have control over which actions can be done anonymously, make sure that at least *all* read-only actions are permitted to non-logged-in visitors, and if possible that data entry portals, such as the bug tracker, that tend to bring information from users to developers, can also be used anonymously, although of course anti-spam techniques, such as captchas, may still be necessary.

Mailing Lists / Message Forums

Discussion forums in which participants post and respond to messages are the bread and butter of project communications. For a long time these were mainly email-based discussion lists, but the distinction between Web-based forums and mailing lists is, thankfully, slowly disappearing. Services like Google Groups [https://groups.google.com/] (which is not itself open source) and Gmane.org [http://Gmane.org/] (which is) have now established that cross-accessibility of message forums as mailing lists and vice versa is the minimum bar to meet, and modern discussion management systems like GroupServer and Sympa reflect this.

Because of this nearly-completed unification between email lists and web-based forums[4], I will use the terms *message forum* and *mailing list* more or less interchangeably. They refer to any kind of message-based forum where posts are linked together in threads (topics), people can subscribe, archives of past messages can be browsed, and the forum can be interacted with via email or via a web browser.

[3]The exception to this is proprietary Javascript code that is received from the hosting site and run confined or "sandboxed" in one tab in the user's browser. The question of whether such code is conceptually an extension of the server, or should be thought of as running on the client machine even though in some senses it has more access to server resources than to client resources, is a deep and ongoing debate. We won't settle it here, but it's at least more complex than just which CPU is executing the instructions.

[4]Which was a long time coming — see rants.org/2008/03/06/thread_theory [http://www.rants.org/2008/03/06/thread_theory/] for more. And no, I'm not too dignified to refer to my own blog post.

If a user is exposed to any channel besides a project's web pages, it is most likely to be one of the project's message forums. But before she experiences the forum itself, she will experience the process of finding the right forums. Your project should have a prominently-placed description of all the available public forums, to give newcomers guidance in deciding which ones to browse or post to first. A typical such description might say something like this:

```
The mailing lists are the main day-to-day communication channels for
the Scanley community.  You don't have to be subscribed to post to a
list, but if it's your first time posting (whether you're subscribed
or not), your message may be held in a moderation queue until a
human moderator has a chance to confirm that the message is not spam.
We're sorry for this delay; blame the spammers who make it necessary.

Scanley has the following lists:

users {_AT_} scanley.org:
  Discussion about using Scanley or programming with the Scanley
  API, suggestions of possible improvements, etc.  You can browse the
  users@ archives at <<<link to archive>>> or subscribe here:
  <<<link to subscribe>>>.

dev {_AT_} scanley.org:
  Discussion about developing Scanley.  Maintainers and contributors
  are subscribed to this list.  You can browse the dev@ archives at
  <<<link to archive>>> or subscribe here: <<<link to subscribe>>>.

  (Sometimes threads cross over between users@ and dev@, and
  Scanley's developers will often participate in discussions on both
  lists.  In general if you're unsure where a question or post
  should go, start it out on users@.  If it should be a
  development discussion, someone will suggest moving it over to
  dev@.)

announcements {_AT_} scanley.org:
  This is a low-traffic, subscribe-only list.  The Scanley
  developers post announcements of new releases and occasional other
  news items of interest to the entire Scanley community here, but
  followup discussion takes place on users@ or dev@.
  <<<link to subscribe>>>.

notifications {_AT_} scanley.org:
  All code commit messages, bug tracker tickets, automated
  build/integration failures, etc, are sent to this list.  Most
  developers should subscribe: <<<link to subscribe>>>.

There is also a non-public list you may need to send to, although
only developers are subscribed:

security {_AT_} scanley.org:
  Where the Scanley project receives confidential reports of
  security vulnerabilities.  Of course, the report will be made
  public eventually, but only after a fix is released; see our
  security procedures page for more [...]
```

Choosing the Right Forum Management Software

It's worth investing some time in choosing the right mailing list management system for your project. Modern list management tools offer at least the following features:

Both email- and web-based access

Users should be able to subscribe to the forums by email, and read them on the web (where they are organized into conversations or "threads", just as they would be in a mailreader).

Moderation features

To "moderate" is to check posts, especially first-time posts, to make sure they are not spam before they go out to the entire list. Moderation necessarily involves human administrators, but software can do a great deal to make it easier on the moderators. There is more said about moderation in the section called “Spam Prevention” later in this chapter.

Rich administrative interface

There are many things administrators need to do besides spam moderation — for example, removing obsolete addresses, a task that can become urgent when a recipient's address starts sending "I am no longer at this address" bounces back to the list in response to every list post (though some systems can even detect this and unsubscribe the person automatically). If your forum software doesn't have decent administrative capabilities, you will quickly realize it, and should consider switching to software that does.

Header manipulation

Some people have sophisticated filtering and replying rules set up in their mail readers, and rely on the forum adding or manipulating certain standard headers. See the section called “Identification and Header Management” later in this chapter for more on this.

Archiving

All posts to the managed lists are stored and made available on the web (see the section called “Conspicuous Use of Archives” in Chapter 6, *Communications* for more on the importance of public archives). Usually the archiver is a native part of the message forum system; occasionally, it is a separate tool that needs to be integrated.

The point of the above list is really just to show that forum management is a complex problem that has already been given a lot of thought, and to some degree been solved. You don't need to become an expert, but you will have to learn at least a little bit about it, and you should expect list management to occupy your attention from time to time in the course of running any free software project. Below we'll examine a few of the most common issues.

Spam Prevention

Between when this sentence is written and when it is published, the Internet-wide spam problem will probably double in severity—or at least it will feel that way. There was a time, not so long ago, when one could run a mailing list without taking any spam-prevention measures at all. The occasional stray post would still show up, but infrequently enough to be only a low-level annoyance. That era is gone forever. Today, a mailing list that takes no spam prevention measures will quickly be submerged in junk emails, to the point of unusability. Spam prevention is mandatory.

We divide spam prevention into two categories: preventing spam posts from appearing on your mailing lists, and preventing your mailing list from being a source of new email addresses for spammers' harvesters. The former is more important to your project, so we examine it first.

Filtering posts

There are three basic techniques for preventing spam posts, and most mailing list software offers all three. They are best used in tandem:

1. **Only auto-allow postings from list subscribers.**

   This is effective as far as it goes, and also involves very little administrative overhead, since it's usually just a matter of changing a setting in the mailing list software's configuration. But note that posts which aren't automatically approved must not be simply discarded. Instead, they should go into a moderation queue, for two reasons. First, you want to allow non-subscribers to post: a person with a question or suggestion should not need to subscribe to a mailing list just to ask a question there. Second, even subscribers may sometimes post from an address other than the one by which they're subscribed. Email addresses are not a reliable method of identifying people, and shouldn't be treated as such.

2. **Filter posts through spam-detection software.**

   If the mailing list software makes it possible (most do), you can have posts filtered by spam-filtering software. Automatic spam-filtering is not perfect, and never will be, since there is a never-ending arms race between spammers and filter writers. However, it can greatly reduce the amount of spam that makes it through to the moderation queue, and since the longer that queue is the more time humans must spend examining it, any amount of automated filtering is beneficial.

   There is not space here for detailed instructions on setting up spam filters. You will have to consult your mailing list software's documentation for that (see the section called “Mailing List / Message Forum Software” later in this chapter). List software often comes with some built-in spam prevention features, but you may want to add some third-party filters. I've had good experiences with SpamAssassin (spamassassin.apache.org [https://spamassassin.apache.org/]) and SpamProbe (spamprobe.sourceforge.net [http://spamprobe.sourceforge.net/]), but this is not a comment on the many other open source spam filters out there, some of which are apparently also quite good. I just happen to have used those two myself and been satisfied with them.

3. **Moderation.**

   For mails that aren't automatically allowed by virtue of being from a list subscriber, and which make it through the spam filtering software, if any, the last stage is *moderation*: the mail is routed to a special holding area, where a human examines it and confirms or rejects it.

   Confirming a post usually takes one of two forms: you can accept the sender's post just this once, or you can tell the system to allow this and all future posts from the same sender. You almost always want to do the latter, in order to reduce the future moderation burden — after all, someone who has made a valid post to a forum is unlikely to suddenly turn into a spammer later.

   Rejecting is done by either marking the item to be discarded, or by explicitly telling the system the message was spam so the system can improve its ability to recognize future spams. Sometimes you also have the option to automatically discard future mails from the same sender without them ever being held in the moderation queue, but there is rarely any point doing this, since spammers don't send from the same address twice anyway.

   Oddly, most message-forum systems have not yet given the moderation queue administrative interface the attention it deserves, considering how common the task is, so moderation often still requires more clicks and UI gestures than it should. I hope this situation will improve in the future. In the meantime, perhaps knowing you're not alone in your frustration will temper your disappointment somewhat.

Be sure to use moderation *only* for filtering out spams, and perhaps for clearly off-topic messages such as when someone accidentally posts to the wrong mailing list. Although the moderation system may give you a way to respond directly to the sender, you should never use that method to answer questions that really belong on the mailing list itself, even if you know the answer off the top of your head. To do so would deprive the project's community of an accurate picture of what sorts of questions people are asking, and deprive people of a chance to answer questions themselves and/or see answers from others. Mailing list moderation is strictly about keeping the list free of spam and of wildly off-topic emails, nothing more.

Address hiding in archives

To prevent your mailing lists from being a source of addresses for spammers, a common technique is for the archiving software to obscure people's email addresses, for example by replacing

```
jrandom@somedomain.com
```

with

```
jrandom_AT_somedomain.com
```

or

```
jrandomNOSPAM@somedomain.com
```

or some similarly obvious (to a human) encoding. Since spam address harvesters often work by crawling through web pages—including your mailing list's online archives—and looking for sequences containing "@", encoding the addresses is a way of making people's email addresses invisible or useless to spammers. This does nothing to prevent spam from being sent to the mailing list itself, of course, but it does avoid increasing the amount of spam sent directly to list users' personal addresses.

Address hiding can be controversial. Some people like it a lot, and will be surprised if your archives don't do it automatically. Other people think it's too much of an inconvenience (because humans also have to translate the addresses back before using them). Sometimes people assert that it's ineffective, because a harvester could in theory compensate for any consistent encoding pattern. However, note that there is empirical evidence that address hiding *is* effective; see the CDT report "Why Am I Getting All This Spam? Unsolicited Commercial E-mail Research Six Month Report" [http://www.spamhelp.org/articles/030319spamreport.pdf].

Ideally, the list management software would leave the choice up to each individual subscriber, either through a special yes/no header or a setting in that subscriber's list account preferences. However, I don't know of any software which offers per-subscriber or per-post choice in the matter, so for now the list manager must make a decision for everyone (assuming the archiver offers the feature at all, which is not always the case). For what it's worth, I lean toward turning address hiding on. Some people are very careful to avoid posting their email addresses on web pages or anywhere else a spam harvester might see it, and they would be disappointed to have all that care thrown away by a mailing list archive; meanwhile, the inconvenience address hiding imposes on archive users is very slight, since it's trivial to transform an obscured address back to a valid one if you need to reach the person. But keep in mind that, in the end, it's still an arms race: by the time you read this, harvesters might well have evolved to the point where they can recognize most common forms of hiding, and we'll have to think of something else.

Identification and Header Management

When interacting with the forum by email, subscribers often want to put mails from the list into a project-specific folder, separate from their other mail. Their mail reading software can do this automatically by examining the mail's *headers*. The headers are the fields at the top of the mail that indicate the sender, recipient, subject, date, and various other things about the message. Certain headers are well known and are effectively mandatory:

```
From: ...
To: ...
Subject: ...
Date: ...
```

Others are optional, though still quite standard. For example, emails are not strictly required to have the

```
Reply-to: sender@email.address.here
```

header, but most do, because it gives recipients a foolproof way to reach the author (it is especially useful when the author had to send from an address other than the one to which replies should be directed).

Some mail reading software offers an easy-to-use interface for filing mails based on patterns in the Subject header. This leads people to request that the mailing list add an automatic prefix to all Subjects, so they can set their readers to look for that prefix and automatically file the mails in the right folder. The idea is that the original author would write:

```
Subject: Making the 2.5 release.
```

but the mail would show up on the list looking like this:

```
Subject: [Scanley Discuss] Making the 2.5 release.
```

Although most list management software offers the option to do this, you may decide against turning the option on. The problem it solves can often be solved in less obtrusive ways (see below), and there is a cost to eating space in the Subject field. Experienced mailing list users typically scan the Subjects of the day's incoming list mail to decide what to read and/or respond to. Prepending the list's name to the Subject can push the right side of the Subject off the screen, rendering it invisible. This obscures information that people depend on to decide what mails to open, thus reducing the overall functionality of the mailing list for everyone.

Instead of munging the Subject header, your project could take advantage of the other standard headers, starting with the To header, which should say the mailing list's address:

```
To: <discuss@lists.example.org>
```

Any mail reader that can filter on Subject should be able to filter on To just as easily.

There are a few other optional-but-standard headers expected for mailing lists; they are sometimes not displayed by most mailreader software, but they are present nonetheless. Filtering on them is even more reliable than using the "To" or "Cc" headers, and since these headers are added to each post by the mailing list management software itself, some users may be counting on their presence:

```
list-help: <mailto:discuss-help@lists.example.org>
list-unsubscribe: <mailto:discuss-unsubscribe@lists.example.org>
list-post: <mailto:discuss@lists.example.org>
Delivered-To: mailing list discuss@lists.example.org
Mailing-List: contact discuss-help@lists.example.org; run by ezmlm
```

For the most part, they are self-explanatory. See nisto.com/listspec/list-manager-intro.html [http://www.nisto.com/listspec/list-manager-intro.html] for more explanation, or if you need the really detailed, formal specification, see faqs.org/rfcs/rfc2369.html [http://www.faqs.org/rfcs/rfc2369.html].

Having said all that, these days I find that most subscribers just request that the Subject header include a list-identifying prefix. That's increasingly how people are accustomed to filtering email: Subject-based filtering is what many of the major online email services (like Gmail) offer users by default, and those services tend not to make it easy to see the presence of less-commonly used headers like the ones I mentioned above — thus making it hard for people to figure out that they would even have the option of filtering on those other headers.

Therefore, reluctantly, I recommend using a Subject prefix (keep it as short as you can) if that's what your community wants. But if your project highly technical and most of its participants are comfortable using the other headers, then that option is always there as a more space-efficient alternative.

It also used to be the case that if you have a mailing list named "foo", then you also have administrative addresses "foo-help" and "foo-unsubscribe" available. In addition to these, it was traditional to have "foo-subscribe" for joining, and "foo-owner", for reaching the list administrators. Increasingly, however, subscribers manage their list membership via Web-based interfaces, so even if the list management software you use sets up these administrative addresses, they may go largely unused.

Some mailing list software offers an option to append unsubscription instructions to the bottom of every post. If that option is available, turn it on. It causes only a couple of extra lines per message, in a harmless location, and it can save you a lot of time, by cutting down on the number of people who mail you —or worse, mail the list!—asking how to unsubscribe.

The Great Reply-to Debate

Earlier, in the section called “Avoid Private Discussions”, I stressed the importance of making sure discussions stay in public forums, and talked about how active measures are sometimes needed to prevent conversations from trailing off into private email threads; furthermore, this chapter is all about setting up project communications software to do as much of the work for people as possible. Therefore, if the mailing list management software offers a way to automatically cause discussions to stay on the list, you would think turning on that feature would be the obvious choice.

Well, not quite. There is such a feature, but it has some pretty severe disadvantages. The question of whether or not to use it is one of the hottest debates in mailing list management—admittedly, not a controversy that's likely to make the evening news in your city, but it can flare up from time to time in free software projects. Below, I will describe the feature, give the major arguments on both sides, and make the best recommendation I can.

The feature itself is very simple: the mailing list software can, if you wish, automatically set the Reply-to header on every post to redirect replies to the mailing list. That is, no matter what the original sender puts in the Reply-to header (or even if they don't include one at all), by the time the list subscribers see the post, the header will contain the list address:

```
Reply-to: discuss@lists.example.org
```

On its face, this seems like a good thing. Because virtually all mail reading software pays attention to the Reply-to header, now when anyone responds to a post, their response will be automatically addressed to the entire list, not just to the sender of the message being responded to. Of course, the responder can still manually change where the message goes, but the important thing is that *by default* replies are directed to the list. It's a perfect example of using technology to encourage collaboration.

Unfortunately, there are some disadvantages. The first is known as the *Can't Find My Way Back Home* problem: sometimes the original sender will put their "real" email address in the Reply-to field, because for one reason or another they send email from a different address than where they receive it. People who always read and send from the same location don't have this problem, and may be surprised that it even exists. But for those who have unusual email configurations, or who cannot control how the From

address on their mails looks (perhaps because they send from work and do not have any influence over the IT department), using Reply-to may be the only way they have to ensure that responses reach them. When such a person posts to a mailing list that he's not subscribed to, his setting of Reply-to becomes essential information. If the list software overwrites it[5], he may never see the responses to his post.

The second disadvantage has to do with expectations, and in my opinion is the most powerful argument against Reply-to munging. Most experienced mail users are accustomed to two basic methods of replying: *reply-to-all* and *reply-to-author*. All modern mail reading software has separate keys for these two actions. Users know that to reply to everyone (that is, including the list), they should choose reply-to-all, and to reply privately to the author, they should choose reply-to-author. Although you want to encourage people to reply to the list whenever possible, there are certainly circumstances where a private reply is the responder's prerogative—for example, they may want to say something confidential to the author of the original message, something that would be inappropriate on the public list.

Now consider what happens when the list has overridden the original sender's Reply-to. The responder hits the reply-to-author key, expecting to send a private message back to the original author. Because that's the expected behavior, he may not bother to look carefully at the recipient address in the new message. He composes his private, confidential message, one which perhaps says embarrassing things about someone on the list, and hits the send key. Unexpectedly, a few minutes later his message appears *on the mailing list!* True, in theory he should have looked carefully at the recipient field, and should not have assumed anything about the Reply-to header. But authors almost always set Reply-to to their own personal address (or rather, their mail software sets it for them), and many longtime email users have come to expect that. In fact, when a person deliberately sets Reply-to to some other address, such as the list, she usually makes a point of mentioning this in the body of her message, so people won't be surprised at what happens when they reply.

Because of the possibly severe consequences of this unexpected behavior, my own preference is to configure list management software to never touch the Reply-to header. This is one instance where using technology to encourage collaboration has, it seems to me, potentially dangerous side-effects. However, there are also some powerful arguments on the other side of this debate. Whichever way you choose, you will occasionally get people posting to your list asking why you didn't choose the other way. Since this is not something you ever want as the main topic of discussion on your list, it might be good to have a canned response ready, of the sort that's more likely to stop discussion than encourage it. Make sure you do *not* insist that your decision, whichever it is, is obviously the only right and sensible one (even if you think that's the case). Instead, point out that this is a very old debate, there are good arguments on both sides, no choice is going to satisfy all users, and therefore you just made the best decision you could. Politely ask that the subject not be revisited unless someone has something genuinely new to say, then stay out of the thread and hope it dies a natural death.

Someone may suggest a vote to choose one way or the other. You can do that if you want, but I personally do not feel that counting heads is a satisfactory solution in this case. The penalty for someone who is surprised by the behavior is so huge (accidentally sending a private mail to a public list), and the inconvenience for everyone else is fairly slight (occasionally having to remind someone to respond to the whole list instead of just to you), that it's not clear that the majority, even though they are the majority, should be able to put the minority at such risk.

I have not addressed all aspects of this issue here, just the ones that seemed of overriding importance. For a full discussion, see these two canonical documents, which are the ones people always cite when they're having this debate:

- **Leave Reply-to alone**, *by Chip Rosenthal*

  "'Reply-To' Munging Considered Harmful" [http://www.unicom.com/pw/reply-to-harmful.html]

[5] In theory, the list software could *add* the lists's address to whatever Reply-to destination were already present, if any, instead of overwriting. In practice, for reasons I don't know, most list software overwrites instead of appending.

- **Set Reply-to to list**, *by Simon Hill*

  "Reply-To Munging Considered Useful" [http://www.metasystema.net/essays/reply-to.mhtml]

Despite the mild preference indicated above, I do not feel there is a "right" answer to this question[6], and happily participate in many lists that *do* set Reply-to. The most important thing you can do is settle on one way or the other early, and try not to get entangled in debates about it after that. When the debate re-arises every few years, as it inevitably will, you can point people to the archived discussion from last time.

Two fantasies

Someday, someone will get the bright idea to implement a *reply-to-list* key in a mail reader. It would use some of the custom list headers mentioned earlier to figure out the address of the mailing list, and then address the reply directly to the list only, leaving off any other recipient addresses, since most are probably subscribed to the list anyway. Eventually, other mail readers will pick up the feature, and this whole debate will go away. (Actually, the Mutt [http://www.mutt.org/] mail reader does offer this feature.[7])

An even better solution would be for Reply-to munging to be a per-subscriber preference. Those who want the list to set Reply-to munged (either on others' posts or on their own posts) could ask for that, and those who don't would ask for Reply-to to be left alone. However, I don't know of any list management software that offers this on a per-subscriber basis. For now, we seem to be stuck with a global setting.[8]

Archiving

The technical details of setting up mailing list archiving are specific to the software that's running the list, and are beyond the scope of this book. If you have to choose or configure an archiver, consider these qualities:

Prompt updating

> People will often want to refer to an archived message that was posted recently. If possible, the archiver should archive each post instantaneously, so that by the time a post appears on the mailing list, it's already present in the archives. If that option isn't available, then at least try to set the archiver to update itself every hour or so. (By default, some archivers run their update processes once per night, but in practice that's far too much lag time for an active mailing list.)

Referential stability

> Once a message is archived at a particular URL, it should remain accessible at that exact same URL forever, or as close to forever as possible. Even if the archives are rebuilt, restored from backup, or otherwise fixed, any URLs that have already been made publicly available should remain the same. Stable references make it possible for Internet search engines to index the archives, which is a major boon to users looking for answers. Stable references are also important because mailing list posts and threads are often linked to from the bug tracker (see the section called “Bug Tracker” later in this chapter) or from other project documents.
>
> Ideally, mailing list software would include a message's archive URL, or at least the message-specific portion of the URL, in a header when it distributes the message to recipients. That way peo-

[6] Although there is, of course, a right answer, and it is to leave the original author's Reply-to untouched. RFC 2822 [http://www.ietf.org/rfc/rfc2822.txt], the relevant standards document, says "When the 'Reply-To:' field is present, it indicates the mailbox(es) to which the author of the message suggests that replies be sent.

[7] Shortly after this book appeared, Michael Bernstein [http://www.michaelbernstein.com/] wrote me to say: "There are other email clients that implement a reply-to-list function besides Mutt. For example, Evolution has this function as a keyboard shortcut, but not a button (Ctrl+L)."

[8] Since I wrote that, I've learned that there is at least one list management system that offers this feature: Siesta [http://siesta.unixbeard.net/]. See also this article about it: perl.com/pub/a/2004/02/05/siesta.html [http://www.perl.com/pub/a/2004/02/05/siesta.html]

ple who have a copy of the message would be able to know its archive location without having to actually visit the archives, which would be helpful because any operation that involves one's web browser is automatically time-consuming. Whether any mailing list software actually offers this feature, I don't know; unfortunately, the ones I have used do not. However, it's something to look for (or, if you write mailing list software, it's a feature to consider implementing, please).

Thread support

It should be possible to go from any individual message to the *thread* (group of related messages) that the original message is part of. Each thread should have its own URL too, separate from the URLs of the individual messages in the thread.

Searchability

An archiver that doesn't support searching—on the bodies of messages, as well as on authors and subjects—is close to useless. Note that some archivers support searching by simply farming the work out to an external search engine such as Google [https://www.google.com/]. This is acceptable, but direct search support is usually more fine-tuned, because it allows the searcher to specify that the match must appear in a subject line versus the body, for example.

The above is just a technical checklist to help you evaluate and set up an archiver. Getting people to actually *use* the archiver to the project's advantage is discussed in later chapters, in particular the section called “Conspicuous Use of Archives”.

Mailing List / Message Forum Software

Here are some tools for running message forums. If the site where you're hosting your project already has a default setup, then you can just use that and avoid having to choose. But if you need to install one yourself, below are some possibilities. (Of course, there are probably other tools out there that I just didn't happen to find, so don't take this as a complete list).

- **Discourse** — discourse.org [https://discourse.org/]

  Discourse was built to be the One True Discussion System for Web and mobile, and so far it seems to be living up to its promise. It is open source, supports both browser-based and email-based participation in discussions, and is under active development with commercial support available. You can purchase hosted discourse if you don't want to set up yourself.

- **Google Groups** — groups.google.com [https://groups.google.com/]

  Listing Google Groups here was a tough call. The service is not itself open source, and a few of its administrative functions can be a bit hard to use. However, its advantages are substantial: your group's archives are always online and searchable; you don't have to worry about scalability, backups, or other run-time infrastructure issues; the moderation and spam-prevention features are pretty good (with the latter constantly being improved, which is important in the neverending spam arms race); and Google Groups are easily accessible via both email and web, in ways that are likely to be already familiar to many participants. These are strong advantages. If you just want to get your project started, and don't want to spend too much time thinking about what message forum software or service to use, Google Groups is a good default choice.

- **GroupServer** — http://www.groupserver.org/

  Has built-in archiver and integrated Web-based interface. GroupServer is a bit of work to set up, but once you have it up and running it offers users a good experience. You may able to find free or low-cost hosted GroupServer hosting for your project's forums, for example from OnlineGroups.net [https://OnlineGroups.net/].

- **Sympa** — sympa.org [https://www.sympa.org/]

  Developed and maintained by a consortium of French universities, and designed for a given instance to handle both very large lists (> 700000 members, they claim) and a large number of lists. Sympa can work with a variety of dependencies; for example, you can run it with sendmail, postfix, qmail or exim as the underlying message transfer agent. It has built-in Web-based archiving.

- **Mailman** — list.org [http://www.list.org/]

  For many years, Mailman was the standard for open source project mailing lists. It comes with a built-in archiver, Pipermail, and hooks for plugging in external archivers. Unfortunately, Mailman is showing its age now, and while it is very reliable in terms of message delivery and other under-the-hood functionality, its administrative interfaces — especially for spam moderation and subscription moderation — are frustrating for those accustomed to the modern Web. As of this writing in late 2013, the long-awaited Mailman 3 was still in development but was about to enter beta-testing; by the time you read this, Mailman 3 may be released, and would be worth a look. It is supposed to solve many of the problems of Mailman 2, and may make Mailman a reasonable choice again.

- **Dada** — dadamailproject.com [http://dadamailproject.com/]

  I've not used Dada myself, but it is actively maintained and, at least from outward appearances, quite spiffy. Note that to use it for participatory lists, as opposed to announcement lists, you apparently need to activate the plug-in "Dada Bridge". Commercial Dada hosting and installation offerings are available, or you can download the code and install it yourself.

Version Control

A *version control system* (or *revision control system*) is a combination of technologies and practices for tracking and controlling changes to a project's files, in particular to source code, documentation, and web pages. If you have never used version control before, the first thing you should do is go find someone who has, and get them to join your project. These days, everyone will expect at least your project's source code to be under version control, and probably will not take the project seriously if it doesn't use version control with at least minimal competence.

The reason version control is so universal is that it helps with virtually every aspect of running a project: inter-developer communications, release management, bug management, code stability and experimental development efforts, and attribution and authorization of changes by particular developers. The version control system provides a central coordinating force among all of these areas. The core of version control is *change management*: identifying each discrete change made to the project's files, annotating each change with metadata like the change's date and author, and then replaying these facts to whoever asks, in whatever way they ask. It is a communications mechanism where a change is the basic unit of information.

This section does not discuss all aspects of using a version control system. It's so all-encompassing that it must be addressed topically throughout the book. Here, we will concentrate on choosing and setting up a version control system in a way that will foster cooperative development down the road.

Version Control Vocabulary

This book cannot teach you how to use version control if you've never used it before, but it would be impossible to discuss the subject without a few key terms. These terms are useful independently of any particular version control system: they are the basic nouns and verbs of networked collaboration, and will be used generically throughout the rest of this book. Even if there were no version control systems in the world, the problem of change management would remain, and these words give us a language for talking about that problem concisely.

If you're comfortably experienced with version control already, you can probably skip this section. If you're not sure, then read through this section at least once. Certain version control terms have gradually changed in meaning since the early 2000s, and you may occasionally find people using them in incompatible ways in the same conversation. Being able to detect that phenomenon early in a discussion can often be helpful.

> **"Version" Versus "Revision"**
>
> The word *version* is sometimes used as a synonym for "revision", but I will not use it that way in this book, because it is too easily confused with "version" in the sense of a version of a piece of software—that is, the release or edition number, as in "Version 1.0". However, since the phrase "version control" is already standard, I will continue to use it as a synonym for "revision control" and "change control". Sorry. One of open source's most endearing characteristics is that it has two words for everything, and one word for every two things.

commit

To make a change to the project; more formally, to store a change in the version control database in such a way that it can be incorporated into future releases of the project. "Commit" can be used as a verb or a noun. For example: "I just committed a fix for the server crash bug people have been reporting on Mac OS X. Jay, could you please review the commit and check that I'm not misusing the allocator there?"

push

To publish a commit to a publicly online repository, from which others can incorporate it into their copy of the project's code. When one says one has pushed a commit, the destination repository is usually implied. Often it is the project's master repository, the one from which public releases are made, but not always.

Note that in some version control systems (e.g., Subversion), commits are automatically and unavoidably pushed up to a predetermined central repository, while in others (e.g., Git, Mercurial) the developer chooses when and where to push commits. Because the former types privilege a particular central repository, they are known as "centralized" version control systems, while the latter are known as "decentralized". In general, decentralized systems are the modern trend, especially for open source projects, which benefit from the peer-to-peer relationship between developers' repositories.

pull

(or "update")

To pull others' changes (commits) into your copy of the project. When pulling changes from a project's mainline development branch (see *branch*), people often say "update" instead of "pull", for example: "Hey, I noticed the indexing code is always dropping the last byte. Is this a new bug?" "Yes, but it was fixed last week—try updating and it should go away."

See also the section called “Pull requests”.

commit message or log message

A bit of commentary attached to each commit, describing the nature and purpose of the commit (both terms are used about equally often; I'll use them interchangeably in this book). Log messages are among the most important documents in any project: they are the bridge between the detailed, highly technical meaning of each individual code changes and the more user-visible world of bug-

fixes, features and project progress. Later in this section, we'll look at ways to distribute them to the appropriate audiences; also, the section called “Codifying Tradition” in Chapter 6, *Communications* discusses ways to encourage contributors to write concise and useful commit messages.

repository

A database in which changes are stored and from which they are published. In centralized version control systems, there is a single, master repository, which stores all changes to the project, and each developer works with a kind of latest summary on her own machine. In decentralized systems, each developer has her own repository, changes can be swapped back and forth between repositories arbitrarily, and the question of which repository is the "master" (that is, the one from which public releases are rolled) is defined purely by social convention, instead of by a combination of social convention and technical enforcement.

clone (see also checkout)

To obtain one's own development repository by making a copy of the project's central repository.

checkout

When used in discussion, "checkout" usually means something like "clone", except that centralized systems don't really clone the full repository, they just obtain a *working copy or working files*. When decentralized systems use the word "checkout", they also mean the process of obtaining working files from a repository, but since the repository is local in that case, the user experience is quite different because the network is not involved.

In the centralized sense, a checkout produces a directory tree called a "working copy" (see below), from which changes may be sent back to the original repository.

working copy or working files

A developer's private directory tree containing the project's source code files, and possibly its web pages or other documents, in a form that allows the developer to edit them. A working copy also contains some version control metadata saying what repository it comes from, what branch it represents, and a few other things. Typically, each developer has her own working copy, from which she edits, tests, commits, pulls, pushes, etc.

In decentralized systems, working copies and repositories are usually colocated anyway, so the term "working copy" is less often used. Developers instead tend to say "my clone" or "my copy" or sometimes "my fork".

revision, *change*, *changeset*, or (again) *commit*

A "revision" is a precisely specified incarnation of the project at a point in time, or of a particular file or directory in the project. These days, most systems also use "revision", "change", "changeset", or "commit" to refer to a set of changes committed together as one conceptual unit, if multiple files were involved, though colloquially most people would refer to changeset 12's effect on file F as "revision 12 of F".

These terms occasionally have distinct technical meanings in different version control systems, but the general idea is always the same: they give a way to speak precisely about exact points in time in the history of a file or a set of files (say, immediately before and after a bug is fixed). For example: "Oh yes, she fixed that in revision 10" or "She fixed that in commit fa458b1fac".

When one talks about a file or collection of files without specifying a particular revision, it is generally assumed that one means the most recent revision(s) available.

diff

A textual representation of a change. A diff shows which lines were changed and how, plus a few lines of surrounding context on either side. A developer who is already familiar with some code can usually read a diff against that code and understand what the change did, and often even spot bugs.

tag or snapshot

A label for a particular state of the project at a point in time. Tags are generally used to mark interesting snapshots of the project. For example, a tag is usually made for each public release, so that one can obtain, directly from the version control system, the exact set of files/revisions comprising that release. Tag names are often things like `Release_1_0`, `Delivery_20130630`, etc.

branch

A copy of the project, under version control but isolated so that changes made to the branch don't affect other branches of the project, and vice versa, except when changes are deliberately "merged" from one branch to another (see below). Branches are also known as "lines of development". Even when a project has no explicit branches, development is still considered to be happening on the "main branch", also known as the "main line" or "*trunk*" or "*master*".

Branches offer a way to keep different lines of development from interfering with each other. For example, a branch can be used for experimental development that would be too destabilizing for the main trunk. Or conversely, a branch can be used as a place to stabilize a new release. During the release process, regular development would continue uninterrupted in the main branch of the repository; meanwhile, on the release branch, no changes are allowed except those approved by the release managers. This way, making a release needn't interfere with ongoing development work. See the section called “Use branches to avoid bottlenecks” later in this chapter for a more detailed discussion of branching.

merge or port

To move a change from one branch to another. This includes merging from the main trunk to some other branch, or vice versa. In fact, those are the most common kinds of merges; it is less common to port a change between two non-trunk branches. See the section called “Singularity of information” for more on change porting.

"Merge" has a second, related meaning: it is what some version control systems do when they see that two people have changed the same file but in non-overlapping ways. Since the two changes do not interfere with each other, when one of the people updates their copy of the file (already containing their own changes), the other person's changes will be automatically merged in. This is very common, especially on projects where multiple people are hacking on the same code. When two different changes *do* overlap, the result is a "conflict"; see below.

conflict

What happens when two people try to make different changes to the same place in the code. All version control systems automatically detect conflicts, and notify at least one of the humans involved that their changes conflict with someone else's. It is then up to that human to *resolve* the conflict, and to communicate that resolution to the version control system.

lock

A way to declare an exclusive intent to change a particular file or directory. For example, "I can't commit any changes to the web pages right now. It seems Alfred has them all locked while he fix-

es their background images." Not all version control systems even offer the ability to lock, and of those that do, not all require the locking feature to be used. This is because parallel, simultaneous development is the norm, and locking people out of files is (usually) contrary to this ideal.

Version control systems that require locking to make commits are said to use the *lock-modify-unlock* model. Those that do not are said to use the *copy-modify-merge* model. An excellent in-depth explanation and comparison of the two models may be found at svnbook.red-bean.com/nightly/en/svn.basic.version-control-basics.html#svn.basic.vsn-models [http://svnbook.red-bean.com/nightly/en/svn.basic.version-control-basics.html#svn.basic.vsn-models]. In general, the copy-modify-merge model is better for open source development, and all the version control systems discussed in this book support that model.

Choosing a Version Control System

If you don't already have a strong opinion about which version control system your project should use, then choose Git (git-scm.com [https://git-scm.com/]), and host your project's repositories at GitHub.com [https://github.com/], which offers unlimited free hosting for open source projects.

Git is by now the *de facto* standard in the open source world, as is hosting one's repositories at GitHub. Because so many developers are already comfortable with that combination, choosing it sends the signal that your project is ready for participants. But Git-at-GitHub is not the only viable combination. Two other reasonable choices of version control system are Mercurial [https://www.mercurial-scm.org/] and Subversion [https://subversion.apache.org/]. Mercurial and Git are both decentralized systems, whereas Subversion is centralized. All three are offered at many different free hosting services; some services even support more than one of them (though GitHub only supports Git, as its name suggests). While some projects host their repositories on their own servers, most just put their repositories on one of the free hosting services, as described in the section called “Canned Hosting”.

There isn't space here for an in-depth exploration of why you might choose something other than Git. If you have a reason to do so, then you already know what that reason is. If you don't, then just use Git (and probably on GitHub). If you find yourself using something other than Git, Mercurial, or Subversion, ask yourself why — because whatever that other version control system is, most other developers won't be familiar with it, and it likely has a smaller and less stable community of support around it than the big three do.

Using the Version Control System

The recommendations in this section are not targeted toward a particular version control system, and should be implementable in any of them. Consult your specific system's documentation for details.

Version everything

Keep not only your project's source code under version control, but also its web pages, documentation, FAQ, design notes, and anything else that people might want to edit. Keep them right with the source code, in the same repository tree. Any piece of information worth writing down is worth versioning—that is, any piece of information that could change. Things that don't change should be archived, not versioned. For example, an email, once posted, does not change; therefore, versioning it wouldn't make sense (unless it becomes part of some larger, evolving document).

The reason to version everything together in one place is so that people only have to learn one mechanism for submitting changes. Often a contributor will start out making edits to the web pages or documentation, and move to small code contributions later, for example. When the project uses the same system for all kinds of submissions, people only have to learn the ropes once. Versioning everything to-

gether also means that new features can be committed together with their documentation updates, that branching the code will branch the documentation too, etc.

Don't keep *generated files* under version control. They are not truly editable data, since they are produced programmatically from other files. For example, some build systems create a file named `configure` based on a template in `configure.in`. To make a change to the `configure`, one would edit `configure.in` and then regenerate; thus, only the template `configure.in` is an "editable file." Just version the templates—if you version the generated files as well, people will inevitably forget to regenerate them when they commit a change to a template, and the resulting inconsistencies will cause no end of confusion.

There are technical exceptions to the rule that all editable data should be kept in the same version control system as the code. For example, a project's bug tracker and its wiki hold plenty of editable data, but usually do not store that data in the main version control system[9]. However, they should still have versioning systems of their own, e.g., the comment history in a bug ticket, and the ability to browse past revisions and view differences between them in a wiki.

Browsability

The project's repository should be browsable on the Web. This means not only the ability to see the latest revisions of the project's files, but to go back in time and look at earlier revisions, view the differences between revisions, read log messages for selected changes, etc.

Browsability is important because it is a lightweight portal to project data. If the repository cannot be viewed through a web browser, then someone wanting to inspect a particular file (say, to see if a certain bugfix had made it into the code) would first have to install version control client software locally, which could turn their simple query from a two-minute task into a half-hour or longer task.

Browsability also implies canonical URLs for viewing a particular change (i.e., a commit), and for viewing the latest revision at any given time without specifying its commit identifier. This can be very useful in technical discussions or when pointing people to documentation or examples. If you tell someone a URL that always points to the latest revision of the a file, or to a particular known version, the communication is completely unambiguous, while avoiding the issue of whether the recipient has an up-to-date working copy of the code themselves.

Some version control systems come with built-in repository-browsing mechanisms, and in any case all hosting sites offer it via their web interfaces. But if you need to install a third-party tool to get repository browsing, do so; it's worth it.

Use branches to avoid bottlenecks

Non-expert version control users are sometimes a bit afraid of branching and merging. If you are among those people, resolve right now to conquer any fears you may have and take the time to learn how to do branching and merging. They are not difficult operations, once you get used to them, and they become increasingly important as a project acquires more developers.

Branches are valuable because they turn a scarce resource—working room in the project's code—into an abundant one. Normally, all developers work together in the same sandbox, constructing the same castle. When someone wants to add a new drawbridge, but can't convince everyone else that it would be an improvement, branching makes it possible for her to copy the castle, take it off to an isolated corner, and try out the new drawbridge design. If the effort succeeds, she can invite the other developers to examine the result (in GitHub-speak, this invitation is known as a "pull request" — see the section called “Pull requests”). If everyone agrees that the result is good, she or someone else can tell the version con-

[9]There are development environments that integrate everything into one unified version control world; see Veracity for an example.

trol system to move ("merge") the drawbridge from the branch version of the castle over to the main version, sometimes called the *master branch*.

It's easy to see how this ability helps collaborative development. People need the freedom to try new things without feeling like they're interfering with others' work. Equally importantly, there are times when code needs to be isolated from the usual development churn, in order to get a bug fixed or a release stabilized (see the section called “Stabilizing a Release” and the section called “Maintaining Multiple Release Lines” in Chapter 7, *Packaging, Releasing, and Daily Development*) without worrying about tracking a moving target. At the same time, people need to be able to review and comment on experimental work, whether it's happening in the master branch or somewhere else. Treating branches as first-class, publishable objects makes all this possible.

Use branches liberally, and encourage others to use them. But also make sure that a given branch is only active for as long as needed. Every active branch is a slight drain on the community's attention. Even those who are not working in a branch still stumble across it occasionally; it enters their peripheral awareness from time to time and draws some attention. Sometimes such awareness is desirable, of course, and commit notices should be sent out for branch commits just as for any other commit. But branches should not become a mechanism for dividing the development community's efforts. With rare exceptions, the eventual goal of most branches should be to merge their changes back into the main line and disappear, as soon as possible.

Singularity of information

Merging has an important corollary: never commit the same change twice. That is, a given change should enter the version control system exactly once. The revision (or set of revisions) in which the change entered is its unique identifier from then on. If it needs to be applied to branches other than the one on which it entered, then it should be merged from its original entry point to those other destinations—as opposed to committing a textually identical change, which would have the same effect in the code, but would make accurate bookkeeping and release management much harder.

The practical effects of this advice differ from one version control system to another. In some systems, merges are special events, fundamentally distinct from commits, and carry their own metadata with them. In others, the results of merges are committed the same way other changes are committed, so the primary means of distinguishing a "merge commit" from a "new change commit" is in the log message. In a merge's log message, don't repeat the log message of the original change. Instead, just indicate that this is a merge, and give the identifying revision of the original change, with at most a one-sentence summary of its effect. If someone wants to see the full log message, she should consult the original revision. Non-duplication makes it easier to be sure when one has tracked down the original source of a change: when you're looking at a complete log message that doesn't refer to a some other merge source, you can know that it must be the original change, and treat it accordingly.

The same principle applies to reverting a change. If a change is withdrawn from the code, then the log message for the reversion should merely state that some specific revision(s) is being reverted, and explain why. It should not describe the semantic code change that results from the reversion, since that can be derived by consulting the original log message and change. (And if you're using a system in which editing past log messages is possible, such as Subversion, go back and edit the original change's log message to mention the future reversion.

All of the above implies that you should use a consistent syntax for referring to changes. This is helpful not only in log messages, but in emails, the bug tracker, and elsewhere. In Git and Mercurial, the syntax is usually "commit bb2377" (where the commit hash code on the right is long enough to be unique in the relevant context); in Subversion, revision numbers are linearly incremented integers and the standard syntax for, say, revision 1729 is "r1729". In other systems, there is usually a standard syntax for expressing the changeset name. Whatever the appropriate syntax is for your system, encourage people to use it when referring to changes. Consistent expression of change names makes project bookkeeping much

easier (as we will see in Chapter 6, *Communications* and Chapter 7, *Packaging, Releasing, and Daily Development*), and since a lot of this bookkeeping may be done by developers who must also use some different bookkeeping method for internal projects at their company, it needs to be as easy as possible.

See also the section called “Releases and Daily Development” in Chapter 7, *Packaging, Releasing, and Daily Development*.

Authorization

Many version control systems offer a feature whereby certain people can be allowed or disallowed from committing in specific sub-areas of the master repository. Following the principle that when handed a hammer, people start looking around for nails, many projects use this feature with abandon, carefully granting people access to just those areas where they have been approved to commit, and making sure they can't commit anywhere else. (See the section called “Committers” in Chapter 8, *Managing Participants* for how projects decide who can put changes where.)

Exercising such tight control is usually unnecessary, and may even be harmful. Some projects simply use an honor system: when a person is granted commit access, even for a sub-area of the project, what they actually receive is the ability to commit anywhere in the master repository. They're just asked to keep their commits in their area. Remember that there is little real risk here: the repository provides an audit trail, and in an active project, all commits are reviewed anyway. If someone commits where they're not supposed to, others will notice it and say something. If a change needs to be undone, that's simple enough—everything's under version control anyway, so just revert.

There are several advantages to this more relaxed approach. First, as developers expand into other areas (which they usually will if they stay with the project), there is no administrative overhead to granting them wider privileges. Once the decision is made, the person can just start committing in the new area right away.

Second, expansion can be done in a more fine-grained manner. Generally, a committer in area X who wants to expand to area Y will start posting patches against Y and asking for review. If someone who already has commit access to area Y sees such a patch and approves of it, she can just tell the submitter to commit the change directly (mentioning the reviewer/approver's name in the log message, of course). That way, the commit will come from the person who actually wrote the change, which is preferable from both an information management standpoint and from a crediting standpoint.

Last, and perhaps most important, using the honor system encourages an atmosphere of trust and mutual respect. Giving someone commit access to a subdomain is a statement about their technical preparedness—it says: "We see you have expertise to make commits in a certain domain, so go for it." But imposing strict authorization controls says: "Not only are we asserting a limit on your expertise, we're also a bit suspicious about your *intentions*." That's not the sort of statement you want to make if you can avoid it. Bringing someone into the project as a committer is an opportunity to initiate them into a circle of mutual trust. A good way to do that is to give them more power than they're supposed to use, then inform them that it's up to them to stay within the stated limits.

The Subversion project has operated on this honor system way or well over a decade, with more than 40 full committers and many more partial committers as of this writing. The only distinction the system actually enforces is between committers and non-committers; further subdivisions are maintained solely by human judgement. Yet the project never had a serious problem with someone deliberately committing outside their domain. Once or twice there's been an innocent misunderstanding about the extent of someone's commit privileges, but it's always been resolved quickly and amiably.

Obviously, in situations where self-policing is impractical, you must rely on hard authorization controls. But such situations are rare. Even when there are millions of lines of code and hundreds or thousands of developers, a commit to any given code module should still be reviewed by those who work on that

module, and they can recognize if someone committed there who wasn't supposed to. If regular commit review *isn't* happening, then the project has bigger problems to deal with than the authorization system anyway.

In summary, don't spend too much time fiddling with the version control authorization system, unless you have a specific reason to. It usually won't bring much tangible benefit, and there are advantages to relying on human controls instead.

None of this should be taken to mean that the restrictions themselves are unimportant, of course. It would be bad for a project to encourage people to commit in areas where they're not qualified. Furthermore, in many projects, full (unrestricted) commit access has a special corollary status: it implies voting rights on project-wide questions. This political aspect of commit access is discussed more in the section called “Who Votes?” in Chapter 4, *Social and Political Infrastructure*.

Receiving and reviewing contributions

These days the primary means by which changes — code contributions, documentation contributions, etc — reach a project is via "pull requests" (described in more detail below), though some older projects still prefer to receive a patch posted to a mailing list or attached in a bug tracker. Once a contribution arrives, it typically goes through a review-and-revise process, involving communication between the contributor and various members of the project. At some point during the process, if all goes well, the contribution is eventually deemed ready for incorporation into the main codebase and is merged in. This does not mean that discussion and work on the contribution cease at that point. The contribution may well continue to be improved, it's just that that improvement now takes place within the project rather than off to one side. The moment when a code change is merged to the project's master branch is when it becomes officially part of the project. It is no longer the sole responsibility of whoever submitted it; it is the collective responsibility of the project as a whole.

Pull requests

A *pull request* is a request *from* a contributor *to* the project that a certain change be "pulled" into the project (usually into the project's master branch, though sometimes pull requests are targeted at some other branch).

The change is offered in the form of the difference between the contributor's copy (or "clone") of the project and the project's own copy. The two copies share most of their change history, of course, but at a certain point the contributor's diverges — it contains the change the contributor has implemented and that the project does not have yet. The project may also have moved on since the clone was made and contain new changes that the contributor does not have, but these can be ignored for the purposes of discussion here. A pull request is directional: it is for sending changes the contributor has that the receiver does not, and is not about changes flowing in the reverse direction.

In practice, the two copies are usually stored on the same hosting site, and the contributor can initiate the pull request by simply clicking a button. On GitHub, and perhaps on other hosting sites, creating a pull request automatically creates a corresponding ticket in the project's bug tracker, so that a pending pull request can be conveniently tracked using the same workflow as any other issue. Some projects have also contributions enter through a collaborative code review tool, such as Gerrit [https://en.wikipedia.org/wiki/Gerrit_%28software%29] or Review Board [https://www.reviewboard.org/], although GitHub has started building some of the features of code-review tools into its pull request management interface.

Pull requests are so frequent a topic of discussion that you will often see people abbreviate them as "PR", as in "Yeah, your proposed fix sounds good. Would you send me a PR please?" For newcomers, however, the term "pull request" is sometimes confusing, however, because it sounds like it is request by the contributor to pull a change from someone else, when actually it is a request the contributor makes to to someone else (the project) to pull the change from the contributor. Some systems use the term *merge*

request to mean the same thing. I actually find that term much more natural, but alas, "pull request" appears to have won, and we all need to just get used to it. I'm not bitter.

Commit notifications / commit emails

Every commit to the repository — or every push containing a group of commits — should generate a notification that goes out to a subscribable forum, such as an email sent to a mailing list. The notification should show who made the change, when they made it, what files and directories changed, and the actual content of the change.

The most common form of commit notifications is just to have a mailing list that developers or any interested party can subscribe to, and send an email to that list for each commit or push. This is a special mailing list devoted to commit emails, separate from the mailing lists to which humans post. Developers should be encouraged to subscribe to the commits list, as it is the most effective way to keep up with what's happening in the project at the code level. Aside from the obvious technical benefits of peer review (see the section called “Practice Conspicuous Code Review”), commit emails help create a sense of community, because they establish a shared environment in which people can react to events that they know are visible to others as well.

Whether your project should use an email list or some other kind of subscribable notification forum depends on the demographics of your developers, but when in doubt, email is usually a good default choice. The specifics of setting up notifications will vary depending on your version control system, but usually there's a script or other packaged facility for doing it. If you're having trouble finding it, try looking for documentation on *hooks* (or sometimes *triggers*) specifically a *post-merge hook* or *post-commit hook* hook. These hooks are a general means of launching automated tasks in response to receiving changes. The hook is fed all the information about the merge, and is then free to use that information to do anything—for example, to send out an email.

With pre-packaged commit email systems, you may want to modify some of the default behaviors:

1. Some commit mailers don't include the actual diffs in the email, but instead provide a URL to view the change on the web using the repository browsing system. While it's good to provide the URL, so the change can be referred to later, it is also important that the commit email include the diffs themselves. Reading email is already part of people's routine, so if the content of the change is visible right there in the commit email, developers will review the commit on the spot, without leaving their mail reader. If they have to click on a URL to review the change, most won't do it, because that requires a new action instead of a continuation of what they were already doing. Furthermore, if the reviewer wants to ask something about the change, it's vastly easier to hit reply-with-text and simply annotate the quoted diff than it is to visit a web page and laboriously cut-and-paste parts of the diff from web browser to email client.

   (Of course, if the diff is huge, such as when a large body of new code has been added to the repository, then it makes sense to omit the diff and offer only the URL. Most commit mailers can do this kind of size-limiting automatically. If yours can't, then it's still better to include diffs, and live with the occasional huge email, than to leave the diffs off entirely. Convenient reviewing and commenting is a cornerstone of cooperative development, and much too important to do without.)

2. The commit emails should set their Reply-to header to the regular development list, not the commit email list. That is, when someone reviews a commit and writes a response, their response should be automatically directed toward the human development list, where technical issues are normally discussed. There are a few reasons for this. First, you want to keep all technical discussion on one list, because that's where people expect it to happen, and because that way there's only one archive to search. Second, there might be interested parties not subscribed to the commit email list. Third, the commit email list advertises itself as a service for watching commits, not for watching commits *and* having occasional technical discussions. Those who subscribed to the commit email list did not sign

up for anything but commit emails; sending them other material via that list would violate an implicit contract.

Note that this advice to set Reply-to does not contradict the recommendations in the section called “The Great Reply-to Debate” earlier in this chapter. It's always okay for the *sender* of a message to set Reply-to. In this case, the sender is the version control system itself, and it sets Reply-to in order to indicate that the appropriate place for replies is the development mailing list, not the commit list.

Bug Tracker

Bug tracking is a broad topic; various aspects of it are discussed throughout this book. Here I'll concentrate mainly on the features your project should look for in a bug tracker, and how to use them. But to get to those, we have to start with a policy question: exactly what kind of information should be kept in a bug tracker?

The term *bug tracker* is misleading. Bug tracking systems are used to track not only bug reports, but new feature requests, one-time tasks, unsolicited patches—really anything that has distinct beginning and end states, with optional transition states in between, and that accrues information over its lifetime. For this reason, bug trackers are also called *issue trackers*, *ticket trackers*, *defect trackers*, *artifact trackers*, *request trackers*, etc.

In this book, I'll generally use the word *ticket* to refer the items in the tracker's database, because that distinguishes between the behavior that the user encountered or proposed — that is, the bug or feature itself — and the tracker's ongoing *record* of that discovery, diagnosis, discussion, and eventual resolution. But note that many projects use the word *bug* or *issue* to refer to both the ticket itself and to the underlying behavior or goal that the ticket is tracking. (In fact, those usages are probably more common than "ticket"; it's just that in this book we need to be able to make that distinction explicitly in a way that projects themselves usually don't.)

The classic ticket life cycle looks like this:

1. Someone files the ticket. They provide a summary, an initial description (including a reproduction recipe, if applicable; see the section called “Treat Every User as a Potential Participant” in Chapter 8, *Managing Participants* for how to encourage good bug reports), and whatever other information the tracker asks for. The person who files the ticket may be totally unknown to the project—bug reports and feature requests are as likely to come from the user community as from the developers.

   Once filed, the ticket is in what's called an *open* state. Because no action has been taken yet, some trackers also label it as *unverified* and/or *unstarted*. It is not assigned to anyone; or, in some systems, it is assigned to a fake user to represent the lack of real assignation. At this point, it is in a holding area: the ticket has been recorded, but not yet integrated into the project's consciousness.

2. Others read the ticket, add comments to it, and perhaps ask the original filer for clarification on some points.

3. The bug gets *reproduced*. This may be the most important moment in its life cycle. Although the bug is not actually fixed yet, the fact that someone besides the original filer was able to make it happen proves that it is genuine, and, no less importantly, confirms to the original filer that they've contributed to the project by reporting a real bug. *(This step and some of the others don't apply to feature proposals, task tickets, etc, of course. But most filings are for genuine bugs, so we'll focus on that here.)*

4. The bug gets *diagnosed*: its cause is identified, and if possible, the effort required to fix it is estimated. Make sure these things get recorded in the ticket; if the person who diagnosed the bug suddenly has to step away from it for a while, someone else should be able to pick up where she left off.

In this stage, or sometimes in the previous one, a developer may "take ownership" of the ticket and *assign* it to herself (the section called “Distinguish clearly between inquiry and assignment” in Chapter 8, *Managing Participants* examines the assignment process in more detail). The ticket's *priority* may also be set at this stage. For example, if it is so important that it should delay the next release, that fact needs to be identified early, and the tracker should have some way of noting it.

5. The ticket gets scheduled for resolution. Scheduling doesn't necessarily mean naming a date by which it will be fixed. Sometimes it just means deciding which future release (not necessarily the next one) the bug should be fixed by, or deciding that it need not block any particular release. Scheduling may also be dispensed with, if the bug is quick to fix.

6. The bug gets fixed (or the task completed, or the patch applied, or whatever). The change or set of changes that fixed it should be discoverable from the ticket. After this, the ticket is *closed* and/or marked as *resolved*.

There are some common variations on this life cycle. Sometimes a ticket is closed very soon after being filed, because it turns out not to be a bug at all, but rather a misunderstanding on the part of the user. As a project acquires more users, more and more such invalid tickets will come in, and developers will close them with increasingly short-tempered responses. Try to guard against the latter tendency. It does no one any good, as the individual user in each case is not responsible for all the previous invalid tickets; the statistical trend is visible only from the developers' point of view, not the user's. (In the section called “Pre-Filtering the Bug Tracker” later in this chapter, we'll look at techniques for reducing the number of invalid tickets.) Also, if different users are experiencing the same misunderstanding over and over, it might mean that aspect of the software needs to be redesigned. This sort of pattern is easiest to notice when there is an issue manager monitoring the bug database; see the section called “Issue Manager” in Chapter 8, *Managing Participants*.

Another common life event for the ticket to be closed as a *duplicate* soon after Step 1. A duplicate is when someone reports something that's already known to the project. Duplicates are not confined to open tickets: it's possible for a bug to come back after having been fixed (this is known as a *regression*), in which case a reasonable course is to reopen the original ticket and close any new reports as duplicates of the original one. The bug tracking system should keep track of this relationship bidirectionally, so that reproduction information in the duplicates is available to the original ticket, and vice versa.

A third variation is for the developers to close the ticket, thinking they have fixed it, only to have the original reporter reject the fix and reopen it. This is usually because the developers simply don't have access to the environment necessary to reproduce the bug, or because they didn't test the fix using the exact same reproduction recipe as the reporter.

Aside from these variations, there may be other small details of the life cycle that vary depending on the tracking software. But the basic shape is the same, and while the life cycle itself is not specific to open source software, it has implications for how open source projects use their bug trackers.

The tracker is as much a public face of the project as the repository, mailing lists or web pages[10]. Anyone may file a ticket, anyone may look at a ticket, and anyone may browse the list of currently open tickets. It follows that you never know how many people are waiting to see progress on a given ticket. While the size and skill of the development community constrains the rate at which tickets can be resolved, the project should at least try to acknowledge each ticket the moment it appears. Even if the ticket lingers for a while, a response encourages the reporter to stay involved, because she feels that a human has registered what she has done (remember that filing a ticket usually involves more effort than, say, posting an email). Furthermore, once a ticket is seen by a developer, it enters the project's con-

[10]Indeed, as the section called “Evaluating Open Source Projects” in Chapter 5, *Organizations, Money, and Business* discusses, the bug tracker is actually the first place to look, even before the repository, when you're trying to evaluate a project's overall health.

sciousness, in the sense that the developer can be on the lookout for other instances of the ticket, can talk about it with other developers, etc.

This centrality to the life of the project implies a few things about trackers' technical features:

- The tracker should be connected to email, such that every change to a ticket, including its initial filing, causes a notification mail to go out to some set of appropriate recipients. See the section called “Interaction with Email” later in this chapter for more on this.

- The form for filing tickets should have a place to record the reporter's email address or other contact information, so she can be contacted for more details. But if possible, it should not *require* the reporter's email address or real identity, as some people prefer to report anonymously. See the section called “Anonymity and involvement” later in this chapter for more on the importance of anonymity.

- The tracker should have APIs. I cannot stress the importance of this enough. If there is no way to interact with the tracker programmatically, then in the long run there is no way to interact with it scalably. APIs provide a route to customizing the behavior of the tracker by, in effect, expanding it to include third-party software. Instead of being just the specific ticket tracking software running on a server somewhere, it's that software *plus* whatever custom behaviors your project implements elsewhere and plugs in to the tracker via the APIs.

  Also, if your project uses a proprietary ticket tracker, as is becoming more common now that so many projects host their code on proprietary-but-free-of-charge hosting sites and just use the site's built-in tracker, APIs provide a way to avoid being locked in to that hosting platform. You can, in theory, take the ticket history with you if you choose to go somewhere else (you may never exercise this option, but think of it as insurance — and some projects have actually done it).

  Currently, the ticket trackers of the big three hosting sites (GitHub, Google Code Hosting, and SourceForge) all have APIs, fortunately. Of them, only SourceForge is itself open source, running a platform called *Allura*[11].

Interaction with Email

Most trackers now have at least decent email integration features: at a minimum, the ability to create new tickets by email, the ability to "subscribe" to a ticket to receive emails about activity on that ticket, and the ability to add new comments to a ticket by email. Some trackers even allow one to manipulate ticket state (e.g., change the status field, the assignee, etc) by email, and for people who use the tracker a lot, such as an the section called “Issue Manager”, that can make a huge difference in their ability to stay on top of tracker activity and keep things organized.

The tracker email feature that is likely to be used by everyone, though, is simply the ability to read a ticket's activity by email and respond by email. This is a valuable time-saver for many people in the project, since it makes it easy to integrate bug traffic into one's daily email flow. But don't let this integration give anyone the illusion that the total collection of bug tickets and their email traffic is the equivalent of the development mailing list. It's not, and the section called “Choose the Right Forum” in Chapter 6, *Communications* discusses why this is important and how to manage the difference.

Pre-Filtering the Bug Tracker

Most ticket databases eventually suffer from the same problem: a crushing load of duplicate or invalid tickets filed by well-meaning but inexperienced or ill-informed users. The first step in combatting this

[11]Oddly, SourceForge's API was also the hardest to find documentation for, though it helps once you know the platform's name is "Allura". For reference, their API documentation is here: sourceforge.net/p/forge/documentation/Allura%20API [https://sourceforge.net/p/forge/documentation/Allura%20API/]

trend is usually to put a prominent notice on the front page of the bug tracker, explaining how to tell if a bug is really a bug, how to search to see if it's already been reported, and finally, how to effectively report it if one still thinks it's a new bug.

This will reduce the noise level for a while, but as the number of users increases, the problem will eventually come back. No individual user can be blamed for it. Each one is just trying to contribute to the project's well-being, and even if their first bug report isn't helpful, you still want to encourage them to stay involved and file better tickets in the future. In the meantime, though, the project needs to keep the ticket database as free of junk as possible.

The two things that will do the most to prevent this problem are: making sure there are people watching the bug tracker who have enough knowledge to close tickets as invalid or duplicates the moment they come in, and requiring (or strongly encouraging) users to confirm their bugs *with other people* before filing them in the tracker.

The first technique seems to be used universally. Even projects with huge ticket databases (say, the Debian bug tracker at bugs.debian.org [https://bugs.debian.org/], which contained 818,841 tickets as of this writing) still arrange things so that *someone* sees each ticket that comes in. It may be a different person depending on the category of the ticket. For example, the Debian project is a collection of software packages, so Debian automatically routes each ticket to the appropriate package maintainers. Of course, users can sometimes misidentify a ticket's category, with the result that the ticket is sent to the wrong person initially, who may then have to reroute it. However, the important thing is that the burden is still shared—whether the user guesses right or wrong when filing, ticket watching is still distributed more or less evenly among the developers, so each ticket is able to receive a timely response.

The second technique is less widespread, probably because it's harder to automate. The essential idea is that every new ticket gets "buddied" into the database. When a user thinks he's found a problem, he is asked to describe it on one of the mailing lists, or in an IRC channel, and get confirmation from someone that it is indeed a bug. Bringing in that second pair of eyes early can prevent a lot of spurious reports. Sometimes the second party is able to identify that the behavior is not a bug, or is fixed in recent releases. Or she may be familiar with the symptoms from a previous ticket, and can prevent a duplicate filing by pointing the user to the older ticket. Often it's enough just to ask the user "Did you search the bug tracker to see if it's already been reported?" Many people simply don't think of that, yet are happy to do the search once they know someone's *expecting* them to.

The buddy system can really keep the ticket database clean, but it has some disadvantages too. Many people will file solo anyway, either through not seeing, or through disregarding, the instructions to find a buddy for new tickets. Thus it is still necessary for some experienced participants to watch the ticket database. Furthermore, because most new reporters don't understand how difficult the task of maintaining the ticket database is, it's not fair to chide them too harshly for ignoring the guidelines. Thus the watchers must be vigilant, and yet exercise restraint in how they bounce unbuddied tickets back to their reporters. The goal is to train each reporter to use the buddying system in the future, so that there is an ever-growing pool of people who understand the ticket-filtering system. On seeing an unbuddied ticket, the ideal steps are:

1. Immediately respond to the ticket, politely thanking the user for filing, but pointing them to the buddying guidelines (which should, of course, be prominently posted on the web site).

2. If the ticket is clearly valid and not a duplicate, approve it anyway, and start it down the normal life cycle. After all, the reporter's now been informed about buddying, so there's no point closing a valid ticket and wasting the work done so far.

3. Otherwise, if the ticket is not clearly valid, close it, but ask the reporter to reopen it if they get confirmation from a buddy. When they do, they should put a reference to the confirmation thread (e.g., a URL into the mailing list archives).

Remember that although this system will improve the signal/noise ratio in the ticket database over time, it will never completely stop the misfilings. The only way to prevent misfilings entirely is to close off the bug tracker to everyone but developers—a cure that is almost always worse than the disease. It's better to accept that cleaning out invalid tickets will always be part of the project's routine maintenance, and to try to get as many people as possible to help.

See also the section called “Issue Manager” in Chapter 8, *Managing Participants*.

IRC / Real-Time Chat Systems

Many projects offer real-time chat rooms using *Internet Relay Chat* (*IRC*), forums where users and developers can ask each other questions and get instant responses. IRC has been around for a long time, and its primarily text-based interface and command language can look old-fashioned — but don't be fooled: the number of people using IRC continues to grow[12], and it is a key communications forum for many open source projects. It's generally the only place where developers can meet in a shared space for real-time conversation on a regular basis.

Alternatives to IRC

Starting in 2015, some newer, primarily web-based real-time group chat systems began gaining momentum, in particular the open source Mattermost [https://mattermost.org/] and the proprietary Slack [https://slack.com/] and Ryver [https://ryver.com/] team communications services. Mattermost is increasingly popular with open source projects, and offers gateways to IRC and other older protocols (so does Slack; I don't know about Ryver, but if they do offer it they're at least not bragging about it on their web pages).

It's too early to know how all this will shake out in the long run, but IRC is still a good choice and probably will continue to be what most open source projects use for some time. If you think IRC won't be suitable for your development community, look at Mattermost next; apparently hosted service will be available at Mattermost.com [https://mattermost.com/] soon, though it wasn't ready as of this writing in early 2016.

If you've never used IRC before, don't be daunted. It's not hard; although there isn't space in this book for an IRC primer, irchelp.org [http://irchelp.org/] is a good guide to IRC usage and administration, and in particular see the tutorial at irchelp.org/irchelp/irctutorial.html [http://www.irchelp.org/irchelp/irctutorial.html]. While in theory your project *could* run its own IRC servers, it is generally not worth the hassle. Instead, just do what everyone else does: host your project's IRC channels[13] at Freenode (freenode.net [https://freenode.net/]). Freenode gives you the control you need to administer your project's IRC channels, while sparing you the not-insignificant trouble of maintaining an IRC server yourself.

The first thing to do is choose a channel name. The most obvious choice is the name of your project—if that's available at Freenode, then use it. If not, try to choose something as close to your project's name, and as easy to remember, as possible. Advertise the channel's availabity from your project's web site, so a visitor with a quick question will see it right away.[14]. If your project's channel gets too noisy, you can divide into multiple channels, for example one for installation problems, another for usage questions, an-

[12] See freenode.net/history.shtml [https://freenode.net/history.shtml] for example.

[13] An IRC *channel* is a single "chat room" — a shared space in which people can "talk" to each other using text. A given IRC server usually hosts many different channels. When a user connects to the server, she chooses which of those channels to join, or her client software remembers and auto-joins them for her. To speak to a particular person in an IRC channel, it is standard to address them by their username (*nickname* or *nick*), so they can pick out your inquiry from the other conversation in the room; see rants.org/2013/01/09/the-irc-curmudgeon [http://www.rants.org/2013/01/09/the-irc-curmudgeon/] for more on this practice.

[14] In fact, you can even offer an IRC chat portal right on your web site. See webchat.freenode.net [https://webchat.freenode.net/] — from the dropdown menu in the upper left corner, choose "Add webchat to your site" and follow the instructions.

other for development chat, etc (the section called “Handling Growth” in Chapter 6, *Communications* discusses when and how to divide into multiple channels). But when your project is young, there should only be one channel, with everyone talking together. Later, as the user-to-developer ratio increases, separate channels may become necessary.

How will people know all the available channels, let alone which channel to talk in? And when they talk, how will they know what the local conventions are?

The answer is to tell them by setting the *channel topic*.[15] The channel topic is a brief message each user sees when they first enter the channel. It gives quick guidance to newcomers, and pointers to further information. For example:

```
The Apache (TM) Subversion (R) version control system
(http://subversion.apache.org/) | Don't ask to ask; just ask your
question! | Read the book: http://www.svnbook.org/ | No one here? Try
http://subversion.apache.org/mailing-lists |
http://subversion.apache.org/faq | Subversion 1.8.8 and 1.7.16 released
```

That's terse, but it tells newcomers what they need to know. It says exactly what the channel is for, gives the project home page (in case someone wanders into the channel without having first been to the project web site), gives a pointer to some documentation, and gives recent release news.

Paste Sites

An IRC channel is a shared space: everyone can see what everyone else is saying. Normally, this is a good thing, as it allows people to jump into a conversation when they think they have something to contribute, and allows spectators to learn by watching. But it becomes problematic when someone has to provide a large quantity of information at once, such as a large error message or a transcript from a debugging session, because pasting too many lines of output into the room will disrupt other conversations.

The solution is to use one of the *pastebin* or *pastebot* sites. When requesting a large amount of data from someone, ask them not to paste it into the channel, but instead to go to (for example) pastebin.ca [https://pastebin.ca/], paste their data into the form there, and tell the resulting new URL to the IRC channel. Anyone can then visit the URL and view the data.

There are many free paste sites available, far too many for a comprehensive list. Three that I seen used a lot are GitHub Gists (gist.github.com [https://gist.github.com/]), paste.lisp.org [http://paste.lisp.org/] and pastebin.ca [https://pastebin.ca/]. But there are many other fine ones, and it's okay if different people in your IRC channel choose to use different paste sites.

IRC Bots

Many technically-oriented IRC channels have a non-human member, a so-called *bot*, that is capable of storing and regurgitating information in response to specific commands. Typically, the bot is addressed just like any other member of the channel, that is, the commands are delivered by "speaking to" the bot. For example:

```
<kfogel> wayita: learn diff-cmd = http://subversion.apache.org/faq.html#diff-cmd
<wayita> Thanks!
```

[15] To set a channel topic, use the `/topic` command. All commands in IRC start with "`/`".

That told the bot, who is logged into the channel as wayita, to remember a certain URL as the answer to the query "diff-cmd" (wayita responded, confirming with a "Thanks!"). Now we can address wayita, asking the bot to tell another user about diff-cmd:

```
<kfogel> wayita: tell jrandom about diff-cmd
<wayita> jrandom: http://subversion.apache.org/faq.html#diff-cmd
```

The same thing can be accomplished via a convenient shorthand:

```
<kfogel> !a jrandom diff-cmd
<wayita> jrandom: http://subversion.apache.org/faq.html#diff-cmd
```

The exact command set and behaviors differ from bot to bot (unfortunately, the diversity of IRC bot command languages seems to be rivaled only by the diversity of wiki syntaxes). The above example happens to be with `wayita` (repos.borg.ch/svn/wayita/trunk [http://repos.borg.ch/svn/wayita/trunk/]), of which there is usually an instance running in `#svn` at Freenode, but there are many other IRC bots available. Note that no special server privileges are required to run a bot. A bot is just like any other user joining a channel.

If your channel tends to get the same questions over and over, I highly recommend setting up a bot. Only a small percentage of channel users will acquire the expertise needed to manipulate the bot, but those users will answer a disproportionately high percentage of questions, because the bot enables them to respond so much more efficiently.

Commit Notifications in IRC

You can also configure a bot to watch your project's version control repository and broadcast commit activity to the relevant IRC channels. Though of somewhat less technical utility than commit emails, since observers might or might not be around when a commit notice pops up in IRC, this technique is of immense *social* utility. People get the sense of being part of something alive and active, and feel that they can see progress being made right before their eyes. And because the notifications appear in a shared space, people in the chat room will often react in real time, reviewing the commit and commenting on it on the spot. The technical details of setting this up are beyond the scope of this book, but it's usually worth the effort. This service used to be provided in an easy-to-use way by the much-missed cia.vc [http://cia.vc/], which shut down in 2011, but several replacements are available: Notifico (n.tkte.ch [https://n.tkte.ch/]), Irker (catb.org/esr/irker [http://www.catb.org/esr/irker/]), and KGB (kgb.alioth.debian.org [https://kgb.alioth.debian.org/]).

Archiving IRC

Although it is possible to publicly archive everything that happens in an IRC channel, it's not necessarily expected. IRC conversations are nominally public, but many people think of them as informal and ephemeral conversations. Users may be careless with grammar, and often express opinions (for example, about other software or other programmers) that they wouldn't want preserved forever in a searchable online archive. Of course, there will sometimes be *excerpts* that get quoted elsewhere, and that's fine. But indiscriminate public logging may make some users uneasy. If you do archive everything, make sure you state so clearly in the channel topic, and give a URL to the archive.

Wikis

When open source software project wikis go bad, they usually go bad for the same reasons: lack of consistent organization and editing, leading to a mess of outdated and redundant pages, and lack of clarity on who the target audience is for a given page or section.

A well-run wiki can be a wonderful thing for users, however, and is worth some effort to maintain. Try to have a clear page organization strategy and even a pleasing visual layout, so that visitors (i.e., potential editors) will instinctively know how to fit their contributions in. Make sure the intended audience is clear at all times to all editors. Most importantly, document these standards in the wiki itself and point people to them, so editors have somewhere to go for guidance. Too often, wiki administrators fall victim to the fantasy that because hordes of visitors are individually adding high quality content to the site, the sum of all these contributions must therefore also be of high quality. That's not how collaborative editing works. Each individual page or paragraph may be good when considered by itself, but it will not be good if embedded in a disorganized or confusing whole.

In general, wikis will amplify any failings that are present from early on, since contributors tend to imitate whatever patterns they see in front of them. So don't just set up the wiki and hope everything falls into place. You must also prime it with well-written content, so people have a template to follow.

The shining example of a well-run wiki is Wikipedia, of course, and in some ways it makes a poor example because it gets so much more editorial attention than any other wiki in the world. Still, if you examine Wikipedia closely, you'll see that its administrators laid a *very* thorough foundation for cooperation. There is extensive documentation on how to write new entries, how to maintain an appropriate point of view, what sorts of edits to make, what edits to avoid, a dispute resolution process for contested edits (involving several stages, including eventual arbitration), and so forth. They also have authorization controls, so that if a page is the target of repeated inappropriate edits, they can lock it down until the problem is resolved. In other words, they didn't just throw some templates onto a web site and hope for the best. Wikipedia works because its founders give careful thought to getting thousands of strangers to tailor their writing to a common vision. While you may not need the same level of preparedness to run a wiki for a free software project, the spirit is worth emulating.

Wikis and Spam

Never allow open, anonymous editing on your wiki. The days when that was possible are *long* gone now; today, any open wiki other than Wikipedia will be covered completely with spam in approximately 3 milliseconds. (Wikipedia is an exception because it has an exceptionally large number of readers willing to clean up spam quickly, and because it has a well-funded organization behind it devoted to resisting spam using various large-scale monitoring techniques not practically available to smaller projects.)

All edits in your project's wiki must come from registered users; if your wiki software doesn't already enforce this by default, then configure it to enforce that. Even then you may need to keep watch for spam edits from users who registered under false pretences for the purpose of spamming.

Choosing a Wiki

If your project is on GitHub or some other free hosting site, it's usually best to use the built-in wiki feature that most such sites offer. That way your wiki will be automatically integrated with your repository or other project permissions, and you can rely on the site's user account system instead of having a separate registration system for the wiki.

If you are setting up your own wiki, then you're free to choose which one, and fortunately there are plenty of good free software wiki implementations available. I've had good experience with DokuWiki (dokuwiki.org/dokuwiki [https://www.dokuwiki.org/dokuwiki]), but there are many others. There is a wonderful tool called the Wiki Choice Wizard at wikimatrix.org [http://www.wikimatrix.org/] that allows you to specify the features you care about (an open source license can be one of them) and then view a chart comparing all the wiki software that meets those criteria. Another good resource is Wikipedia's own list of wikis: en.wikipedia.org/wiki/List_of_wiki_software [https://en.wikipedia.org/wiki/List_of_wiki_software].

I do not recommend using MediaWiki (mediawiki.org [https://www.mediawiki.org]) as the wiki software for most projects. MediaWiki is the software on which Wikipedia itself runs, and while it is very good at that, its administrative facilities are tuned to the needs of a site unlike any other wiki on the Net — and actually not so well-tuned to the needs of smaller editing communities. Many projects are tempted to choose MediaWiki because they think it will be easier for users who already know its editing syntax from having edited at Wikipedia, but this turns out to be an almost non-existent advantage for several reasons. First, wikis in general, including Wikipedia, are tending toward rich-text in-browser editing anyway, so that no one really needs to learn the underlying wiki syntax unless they aim to be a power user. Second, many other wikis offer a MediaWiki-syntax plugin, so you can have that syntax anyway if you really want it. Third, for those who will use a plaintext syntax instead of rich-text editing, it's better to use a standardized generic markup format like Markdown (daringfireball.net/projects/markdown [https://daringfireball.net/projects/markdown/]), which is available in many wikis either natively or via a plugin, than to use a wiki syntax of any flavor. If you support Markdown, then people can edit in your wiki using the same markup syntax they already know from GitHub and other popular tools.

Q&A Forums

In the past few years, online question-and-answer forums (or *Q&A forums*) have gone from being an afterthought offered by the occasional project to an increasingly expected and normal component of user-facing services. A high-quality Q&A forum is like a FAQ with nearly real-time updates — indeed, if your Q&A forum is sufficiently healthy, it often makes sense to either use it directly as your project's FAQ, or have the FAQ consist mostly of pointers to the forum's most popular items.

A project can certainly host its own forums, and many do, using free software such as Askbot [https://askbot.com/] [16], OSQA [http://osqa.net/], Shapado [http://shapado.com/], or Coordino [http://www.coordino.com/]. However, there are also some third-party services that aggregate questions and answers, the best-known of which, stackoverflow.com [https://stackoverflow.com/], frequently has its answers coming up first in generic search engine results for popular questions.

While Stack Overflow hosts Q&A about many things, not just about open source projects, it seems to have found the right combination of cultural guidelines and upvoting/downvoting features to enable its contributors to quickly narrow in on good answers for questions about open source software in particular. (The questions and answers on Stack Overflow are freely licensed, although the code that runs the site itself is not open source.) On the other hand, projects that host their own Q&A forums are lately doing pretty well in search engine results too. It may be that the current dominance of Stack Overflow, as of this writing in 2014, is partly just an accident of timing, and that the real lesson is that Q&A-style forums are an important addition to the free software project communications toolbox — one that scales better with user base than many other tools do.

There is no definite answer to the question of whether or when you should set up dedicated Q&A forums for your project. It depends on available resources, on the type of project, the demographics of the user community, etc. But do keep an eye out for Stack Overflow results, or other third-party results, coming up in generic question-style searches about your project. Their presence may indicate that it's time to consider setting up a dedicated Q&A forum. Whether you do or not, the project can still learn a lot from looking at what people are asking on Stack Overflow, and at the responses.

Translation Infrastructure

In recent years, various platforms have arisen to help automate the organization and integration of human-language translation work in open source projects. "Translation work" here means not just the process of translating the software's documentation, but also its run-time user interface, error messages,

[16] See ask.libreoffice.org [https://ask.libreoffice.org/en/questions/], for example.

etc into different languages, so that each user can interact with the software in their preferred language. (See the section called “Translation Manager” in Chapter 8, *Managing Participants* for more about the translation process.)

It is not strictly necessary to use a separate translation platform at all. Your translators could work directly in the project's repository, like any other developer. But because translation is a specialized skill, and translators' methods are basically the same from project to project, the process is quite amenable to being made more efficient through the use of dedicated tools. Web-based translation platforms make it easier for translators to get involved because such platforms remove the requirement that a translator (who may have linguistic expertise but not development expertise) be comfortable with the project's development tools, and because they are specially optimized for performing translation rather than for performing general code development.

Until 2013, the obvious recommendation for a platform would have been Transifex.com [https://transifex.com/], which was both the premier software translation site and was open source software itself. Transifex is probably still the premier platform in terms of usage, including among open source projects, but its main corporate sponsors forked it into a closed, proprietary version in March 2013[17]. Transifex still offers free hosting for open source projects, but if you prefer that your translators have a fully open source platform to work in, weblate.org [https://weblate.org/] is one, and it offers a commercial hosted service that is free of charge for open source projects as well. Three other open source platforms are Launchpad Translations translations.launchpad.net [https://translations.launchpad.net/], zanata.org [http://zanata.org/], and translatewiki.net [https://translatewiki.net/].

Social Networking Services

Perhaps surprisingly for such social endeavors, open source projects typically make only limited use of what most people think of as "social networking" services. But this seeming omission is really a matter of definition: most of the infrastructure that open source projects have been using for decades, since long before "social networking" became a recognized category of software, is actually social networking software even if it isn't called that. The reason open source projects tend not to have much presence
as projects
on, say, Facebook is just that the services Facebook offers are not well-tuned to what open source projects need. On the other hand, as you might expect, the infrastructure these projects have been using and improving for many years *is* quite well-tuned to their needs.

Most projects do use Twitter and similar microblog services, because sending out short quips and announcements that can be easily forwarded and replied to is a good way for a project to have conversations with its community; see LibreOffice's "@AskLibreOffice" tweet stream at twitter.com/AskLibreOffice [https://twitter.com/AskLibreOffice] for an example of this. Projects also sometimes use services such as Eventbrite.com [https://www.eventbrite.com] and Meetup.com [https://www.Meetup.com/] to arrange in-person meetings of users and developers.

But beyond lightweight services such as those, most free software projects do not maintain a large presence on mainstream social media platforms (though individual developers sometimes do, of course, and they may discuss the project there). The reward the project gets in exchange for that investment of time and attention appears not to be high enough to be worth the effort.

[17] See github.com/transifex/transifex/issues/206#issuecomment-15243207 [https://github.com/transifex/transifex/issues/206#issuecomment-15243207] for more.

Chapter 4. Social and Political Infrastructure

The first questions people usually ask about free software are "How does it work? What keeps a project running? Who makes the decisions?" I'm always dissatisfied with bland responses about meritocracy, the spirit of cooperation, code speaking for itself, etc. The fact is, the question is not easy to answer. Meritocracy, cooperation, and running code are all part of it, but they do little to explain how projects actually run on a day-to-day basis, and say nothing about how conflicts are resolved.

This chapter tries to show the structural underpinnings successful projects have in common. I mean "successful" not just in terms of technical quality, but in terms of operational health and survivability. Operational health is the project's ongoing ability to incorporate new code contributions and new developers, and to be responsive to incoming bug reports. Survivability is the project's ability to exist independently of any individual participant or sponsor—think of it as the likelihood that the project would continue even if all of its founding members were to move on to other things[1]. Technical success is not hard to achieve, but without a robust developer base and social foundation, a project may be unable to handle the growth that initial success brings, or the departure of charismatic individuals.

There are various ways to achieve this kind of success. Some involve a formal governance structure, by which debates are resolved, new developers are invited in (and sometimes out), new features planned, and so on. Others use less formal structure, but more self-restraint, to produce an atmosphere of fairness that people can rely on as a *de facto* form of governance. Both ways lead to the same result: a sense of institutional permanence, supported by habits and procedures that are well understood by everyone who participates. These features are even more important in self-organizing systems than in centrally-controlled ones, because in self-organizing systems, everyone is conscious that a few bad apples can spoil the whole barrel, at least for a while.

Forkability

The indispensable ingredient that binds developers together on a free software project, and makes them willing to compromise when necessary, is the code's *forkability*: the ability of anyone to take a copy of the source code and use it to start a competing project, known as a *fork*. The paradoxical thing is that the *possibility* of forks is usually a much greater force in free software projects than actual forks, which are very rare. Because a fork is usually bad for everyone (for reasons examined in detail in the section called “Forks” in Chapter 8, *Managing Participants*), the more serious the threat of a fork becomes, the more willing people are to compromise to avoid it.

Forks, or rather the potential for forks, are the reason there are no true dictators in free software projects. This may seem like a surprising claim, considering how common it is to hear someone called the "dictator" (sometimes softened to "benevolent dictator") in a given open source project. But this kind of dictatorship is special, quite different from the conventional understanding of the word. Imagine a king whose subjects could copy his entire kingdom at any time and move to the copy to rule as they see fit. Would not such a king govern very differently from one whose subjects were bound to stay under his rule no matter what he did?

This is why even projects that are not formally organized as democracies are, in practice, democracies when it comes to important decisions. Replicability implies forkability; forkability implies consensus. It may well be that everyone is willing to defer to one leader (the most famous example being Linus Torvalds in Linux kernel development), but this is because they *choose* to do so, in an entirely non-cyni-

[1]This is also known as the "*Bus Factor*", that is, how many participants would have to get hit by a bus (figuratively speaking) for the project to be unable to continue. See en.wikipedia.org/wiki/Bus_factor [https://en.wikipedia.org/wiki/Bus_factor].

cal and non-sinister way. The dictator has no magical hold over the project. A key property of all open source licenses is that they do not give one party more power than any other in deciding how the code can be changed or used. If the dictator were to suddenly start making bad decisions, there would be restlessness, followed eventually by revolt and a fork. Except, of course, that things rarely get that far, because the dictator compromises first.

But just because forkability puts an upper limit on how much power anyone can exert in a project doesn't mean there aren't important differences in how projects are governed. You don't want every decision to come down to the last-resort question of who might consider a fork. That would get tiresome very quickly, and sap energy away from real work. The next two sections examine different ways to organize projects such that most decisions go smoothly. These two examples are somewhat idealized extremes; many projects fall somewhere along a continuum between them.

Benevolent Dictators

The *benevolent dictator* model is exactly what it sounds like: final decision-making authority rests with one person, who, by virtue of personality and experience, is expected to use it wisely.

Although "benevolent dictator" (or *BD*)is the standard term for this role, it would be better to think of it as "community-approved arbitrator" or "judge". Generally, benevolent dictators do not actually make all the decisions, or even most of the decisions. It's unlikely that one person could have enough expertise to make consistently good decisions across all areas of the project, and anyway, quality developers won't stay around unless they have some influence on the project's direction. Therefore, benevolent dictators commonly do not dictate much. Instead, they let things work themselves out through discussion and experimentation whenever possible. They participate in those discussions themselves, but as regular developers, often deferring to an area maintainer who has more expertise. Only when it is clear that no consensus can be reached, and that most of the group *wants* someone to guide the decision so that development can move on, does she put her foot down and say "This is the way it's going to be." Reluctance to make decisions by fiat is a trait shared by almost all successful benevolent dictators; it is one of the reasons they manage to keep the role.

Who Can Be a Good Benevolent Dictator?

Being a BD requires a combination of traits. It needs, first of all, a well-honed sensitivity to one's own influence in the project, which in turn brings self-restraint. In the early stages of a discussion, one should not express opinions and conclusions with so much certainty that others feel like it's pointless to dissent. People must be free to air ideas, even stupid ideas. It is inevitable that the BD will post a stupid idea from time to time too, of course, and therefore the role also requires an ability to recognize and acknowledge when one has made a bad decision—though this is simply a trait that *any* good developer should have, especially if she stays with the project a long time. But the difference is that the BD can afford to slip from time to time without worrying about long-term damage to her credibility. Developers with less seniority may not feel so secure, so the BD should phrase critiques or contrary decisions with some sensitivity for how much weight her words carry, both technically and psychologically.

The BD does *not* need to have the sharpest technical skills of anyone in the project. She must be skilled enough to work on the code herself, and to understand and comment on any change under consideration, but that's all. The BD position is neither acquired nor held by virtue of intimidating coding skills. What *is* important is experience and overall design sense—not necessarily the ability to produce good design on demand, but the ability to recognize and endorse good design, whatever its source.

It is common for the benevolent dictator to be a founder of the project, but this is more a correlation than a cause. The sorts of qualities that make one able to successfully start a project—technical competence, ability to persuade other people to join, and so on—are exactly the qualities any BD would need. And of

course, founders start out with a sort of automatic seniority, which can often be enough to make benevolent dictatorship by the founder appear the path of least resistance for all concerned.

Remember that the potential to fork goes both ways. A BD can fork a project just as easily as anyone else, and some have occasionally done so, when they felt that the direction they wanted to take the project was different from where the majority of other developers wanted to go. Because of forkability, it does not matter whether the benevolent dictator has control over the currently accepted "master" project repository. People sometimes talk of repository control as though it were the ultimate source of power in a project, but in fact it is irrelevant. The ability to add or remove people's commit passwords for one project on a particular hosting site affects only that copy of the project on that site. Prolonged abuse of that power, whether by the BD or someone else, would simply lead to development moving to a different copy of the project.

Whether your project should have a benevolent dictator, or would run better with some less centralized form of governance, largely depends on who is available to fill the role. As a general rule, if it's simply obvious to everyone who should be the BD, then that's the way to go. But if no candidate for BD is immediately obvious, then the project should probably use a decentralized decision-making process, as described in the next section.

Consensus-based Democracy

As projects get older, they tend to move away from the benevolent dictatorship model and toward more openly democratic systems. This is not necessarily out of dissatisfaction with a particular BD. It's simply that group-based governance is more "evolutionarily stable", to borrow a biological metaphor. Whenever a benevolent dictator steps down, or attempts to spread decision-making responsibility more evenly, it is an opportunity for the group to settle on a new, non-dictatorial system—establish a constitution, as it were. The group may not take this opportunity the first time, or the second, but eventually they will; once they do, the decision is unlikely ever to be reversed. Common sense explains why: if a group of N people were to vest one person with special power, it would mean that N - 1 people were each agreeing to decrease their individual influence. People usually don't want to do that. Even if they did, the resulting dictatorship would still be conditional: the group anointed the BD, clearly the group could depose the BD. Therefore, once a project has moved from leadership by a charismatic individual to a more formal, group-based system, it rarely moves back.

The details of how these systems work vary widely, but there are two common elements: one, the group works by consensus most of the time; two, there is a formal voting mechanism to fall back on when consensus cannot be reached.

Consensus simply means an agreement that everyone is willing to live with. It is not an ambiguous state: a group has reached consensus on a given question when someone proposes that consensus has been reached and no one contradicts the assertion. The person proposing consensus should, of course, state specifically what the consensus is, and what actions would be taken in consequence of it, if those are not obvious.

Most conversation in a project is on technical topics, such as the right way to fix a certain bug, whether or not to add a feature, how strictly to document interfaces, etc. Consensus-based governance works well because it blends seamlessly with the technical discussion itself. By the end of a discussion, there is often general agreement on what course to take. Someone will usually make a concluding post, which is simultaneously a summary of what has been decided and an implicit proposal of consensus. This provides a last chance for someone else to say "Wait, I didn't agree to that. We need to hash this out some more."

For small, uncontroversial decisions, the proposal of consensus is implicit. For example, when a developer spontaneously commits a bugfix, the commit itself is a proposal of consensus: "I assume we all

agree that this bug needs to be fixed, and that this is the way to fix it." Of course, the developer does not actually say that; she just commits the fix, and the others in the project do not bother to state their agreement, because silence is consent. If someone commits a change that turns out *not* to have consensus, the result is simply for the project to discuss the change as though it had not already been committed. The reason this works is the topic of the next section.

Version Control Means You Can Relax

The fact that the project's source code is kept under version control means that most decisions can be easily unmade. The most common way this happens is that someone commits a change mistakenly thinking everyone would be happy with it, only to be met with objections after the fact. It is typical for such objections to start out with an obligatory apology for having missed out on prior discussion, though this may be omitted if the objector finds no record of such a discussion in the mailing list archives. Either way, there is no reason for the tone of the discussion to be different after the change has been committed than before. Any change can be reverted[2], at least until dependent changes are introduced (i.e., new code that would break if the original change were suddenly removed). The version control system gives the project a way to undo the effects of bad or hasty judgement. This, in turn, frees people to trust their instincts about how much feedback is necessary before doing something.

This also means that the process of establishing consensus need not be very formal. Most projects handle it by feel. Minor changes can go in with no discussion, or with minimal discussion followed by a few nods of agreement. For more significant changes, especially ones with the potential to destabilize a lot of code, people should wait a day or two before assuming there is consensus, the rationale being that no one should be marginalized in an important conversation simply because he didn't check email frequently enough.

Thus, when someone is confident she knows what needs to be done, she should just go ahead and do it. This applies not only to software fixes, but to web site updates, documentation changes, and anything else unlikely to be controversial. Usually there will be only a few instances where an action draws disapproval, and these can be handled on a case-by-case basis. Of course, one shouldn't encourage people to be headstrong. There is still a psychological difference between a decision under discussion and one that has already taken effect but is technically reversible. People always feel that momentum is allied to action, and will be slightly more reluctant to revert a change than to prevent it in the first place. If a developer abuses this fact by committing potentially controversial changes too quickly, however, people can and should complain, and hold that developer to a stricter standard until things improve.

When Consensus Cannot Be Reached, Vote

Inevitably, some debates just won't consense. When all other means of breaking a deadlock fail, the solution is to vote. But before a vote can be taken, there must be a clear set of choices on the ballot. Here, again, the normal process of technical discussion blends serendipitously with the project's decision-making procedures. The kinds of questions that come to a vote often involve complex, multifaceted issues. In any such complex discussion, there are usually one or two people playing the role of *honest broker*: posting periodic summaries of the various arguments and keeping track of where the core points of disagreement (and agreement) lie. These summaries help everyone measure how much progress has been made toward resolving the issues, and remind everyone of what questions remain to be addressed. Those same summaries can serve as prototypes for a ballot sheet, should a vote become necessary. If the honest brokers have been doing their job well, they will be able to credibly call for a vote when the time comes, and the group will be willing to use a ballot sheet based on their summary of the issues. The brokers themselves may be participants in the debate; it is not necessary for them to remain above the fray,

[2]Of course, it's good manners and good sense to discuss before reverting. Reverting a change is not the way to start a conversation about whether it should be reverted. There are sometimes situation where it may be appropriate to perform the reversion before the conversation about it has definitively concluded, but even then it's still important to have started the conversation first.

as long as they can understand and fairly represent others' views, and not let their partisan sentiments prevent them from summarizing the state of the debate accurately.

The actual content of the ballot is usually not controversial By the time matters reach a vote, the disagreement has usually boiled down to a few key issues, with recognizable labels and brief descriptions. Occasionally a developer will object to the form of the ballot itself. Sometimes his concern is legitimate, for example that an important choice was left off or not described accurately. But other times a developer may be merely trying to stave off the inevitable, perhaps knowing that the vote probably won't go his way. See the section called “Difficult People” in Chapter 6, *Communications* for how to deal with this sort of obstructionism.

Remember to specify the voting method, as there are many different kinds, and people might make wrong assumptions about which procedure is being used. A good choice in most cases is *approval voting*[3], whereby each voter can vote for as many of the choices on the ballot as she likes. Approval voting is simple to explain and to count. See en.wikipedia.org/wiki/Voting_system [https://en.wikipedia.org/wiki/Voting_system] for more details about approval voting and other voting systems, but beware the temptation to geek out on voting systems. I did, in the course of researching this paragraph, and I've never been the same since. You can try all sorts of fancy voting methods, for example ones that involve preferential ranking of choices — such as *Borda*, *Condorcet*, *instant runoff*, and *single transferable vote* — but a famous result known as Arrow's impossibility theorem [https://en.wikipedia.org/wiki/Arrow%27s_impossibility_theorem] has already demonstrated that no voting system is perfect. Try to avoid getting into a long debate about which voting system to use (because, of course, you will then find yourself in a debate about which voting system to use to decide the voting system!). Approval voting is a very good choice for the kinds of ballots an open source project is likely to use to resolve technical or procedural questions.

Finally, conduct votes in public as much as possible. There is no need for secrecy or anonymity in a vote on matters that have been debated publicly anyway. Have each participant post her votes to the project mailing list, so that any observer can tally and check the results for herself, and so that everything is recorded in the archives. If you prefer to use specialized software to conduct the vote, as of this writing in mid-2015 the open source voting system Helios (vote.heliosvoting.org [https://vote.heliosvoting.org/]) is a good choice and supports approval voting.

When To Vote

The hardest thing about voting is determining when to do it. In general, taking a vote should be very rare —a last resort for when all other options have failed. Don't think of voting as a great way to resolve debates. It isn't. It ends discussion, and thereby ends creative thinking about the problem. As long as discussion continues, there is the possibility that someone will come up with a new solution everyone likes. This happens surprisingly often: a lively debate can produce a new way of looking at the problem, and lead to a proposal that eventually satisfies everyone. Even when no new proposal arises, it's still usually better to broker a compromise than to hold a vote. After a compromise, everyone is a little bit unhappy, whereas after a vote, some people are unhappy while others are happy. From a political standpoint, the former situation is preferable: at least each person can feel he extracted a price for his unhappiness. He may be dissatisfied, but so is everyone else.

Voting's only function is that it finally settles a question so everyone can move on. But it settles it by a head count, instead of by rational dialogue leading everyone to the same conclusion. The more experienced people are with open source projects, the less eager I find them to be to settle questions by vote. Instead they will try to explore previously unconsidered solutions, or compromise more severely than they'd originally planned. Various techniques are available to prevent a premature vote. The most obvious is simply to say "I don't think we're ready for a vote yet," and explain why not. Another is to ask for an informal (non-binding) show of hands. If the response clearly tends toward one side or another,

[3]Also called *multiple approval*, *multiple preference* or *multiple preference approval.*

this will make some people suddenly more willing to compromise, obviating the need for a formal vote. But the most effective way is simply to offer a new solution, or a new viewpoint on an old suggestion, so that people re-engage with the issues instead of merely repeating the same arguments.

In certain rare cases, everyone may agree that all the compromise solutions are worse than any of the non-compromise ones. When that happens, voting is less objectionable, both because it is more likely to lead to a superior solution and because people will not be overly unhappy no matter how it turns out. Even then, the vote should not be rushed. The discussion leading up to a vote is what educates the electorate, so stopping that discussion early can lower the quality of the result.

(Note that this advice to be reluctant to call votes does not apply to the change-inclusion voting described in the section called “Stabilizing a Release” in Chapter 7, *Packaging, Releasing, and Daily Development*, where voting is more of a communications mechanism, a means of registering one's involvement in the change review process so that everyone can tell how much review a given change has received. It also does not apply to standard procedural elections, for example choosing the board of directors for a project organized as a non-profit legal entity.)

Who Votes?

Having a voting system raises the question of electorate: who gets to vote? This has the potential to be a sensitive issue, because it forces the project to officially recognize some people as being more involved, or as having better judgement, than others.

The best solution is to simply take an existing distinction, commit access, and attach voting privileges to it. In projects that offer both full and partial commit access, the question of whether partial committers can vote largely depends on the process by which partial commit access is granted. If the project hands it out liberally, for example as a way of maintaining many third-party contributed tools in the repository, then it should be made clear that partial commit access is really just about committing, not voting. The reverse implication naturally holds as well: since full committers *will* have voting privileges, they must be chosen not only as programmers, but as members of the electorate. If someone shows disruptive or obstructionist tendencies on the mailing list, the group should be very cautious about making him a committer, even if the person is technically skilled.

The voting system itself should be used to choose new committers, both full and partial. But here is one of the rare instances where secrecy is appropriate. You can't have votes about potential committers posted to a public mailing list, because the candidate's feelings (and reputation) could be hurt. Instead, the usual way is that an existing committer posts to a private mailing list consisting only of the other committers, proposing that someone be granted commit access. The other committers speak their minds freely, knowing the discussion is private. Often there will be no disagreement, and therefore no vote necessary. After waiting a few days to make sure every committer has had a chance to respond, the proposer mails the candidate and offers him commit access. If there is disagreement, discussion ensues as for any other question, possibly resulting in a vote. For this process to be open and frank, the mere fact that the discussion is taking place at all should be secret. If the person under consideration knew it was going on, and then were never offered commit access, he could conclude that he had lost the vote, and would likely feel hurt. Of course, if someone explicitly asks for commit access, then there is no choice but to consider the proposal and explicitly accept or reject him. If the latter, then it should be done as politely as possible, with a clear explanation: "We liked your patches, but haven't seen enough of them yet," or "We appreciate all your patches, but they required considerable adjustments before they could be applied, so we don't feel comfortable giving you commit access yet. We hope that this will change over time, though." Remember, what you're saying could come as a blow, depending on the person's level of confidence. Try to see it from their point of view as you write the mail.

Because adding a new committer is more consequential than most other one-time decisions, some projects have special requirements for the vote. For example, they may require that the proposal receive at least *n* positive votes and no negative votes, or that a supermajority vote in favor. The exact parame-

ters are not important; the main idea is to get the group to be careful about adding new committers. Similar, or even stricter, special requirements can apply to votes to *remove* a committer, though hopefully that will never be necessary. See the section called “Committers” in Chapter 8, *Managing Participants* for more on the non-voting aspects of adding and removing committers.

Polls Versus Votes

For certain kinds of votes, it may be useful to expand the electorate. For example, if the developers simply can't figure out whether a given interface choice matches the way people actually use the software, one solution is to ask to all the subscribers of the project's mailing lists to vote. These are really *polls* rather than votes, but the developers may choose to treat the result as binding. As with any poll, be sure to make it clear to the participants that there's a write-in option: if someone thinks of a better option not offered in the poll questions, her response may turn out to be the most important result of the poll.

Vetoes

Some projects allow a special kind of vote known as a *veto*. A veto is a way for a developer to put a halt to a hasty or ill-considered change, at least long enough for everyone to discuss it more. Think of a veto as somewhere between a very strong objection and a filibuster. Its exact meaning varies from one project to another. Some projects make it very difficult to override a veto; others allow them to be overridden by regular majority vote, perhaps after an enforced delay for more discussion. Any veto should be accompanied by a thorough explanation; a veto without such an explanation should be considered invalid on arrival.

With vetoes comes the problem of veto abuse. Sometimes developers are too eager to raise the stakes of disagreement by casting a veto, when really all that was called for was more discussion. You can prevent veto abuse by being very reluctant to use vetoes yourself, and by gently calling it out when someone else uses her veto too often. If necessary, you can also remind the group that vetoes are binding for only as long as the group agrees they are—after all, if a clear majority of developers wants X, then X is going to happen one way or another. Either the vetoing developer will back down, or the group will decide to weaken the meaning of a veto.

You may see people write "-1" to express a veto. This usage originally comes from the Apache Software Foundation (which has a highly structured voting and veto process, described at apache.org/foundation/voting.html [https://www.apache.org/foundation/voting.html]), but has since spread to many other projects — albeit not always with exactly the same formal meaning as at the ASF. Technically, "-1" does not always indicate a formal veto even according to the Apache standards, but informally it is usually taken to mean a veto, or at least a very strong objection.

Like votes, vetoes can apply retroactively. It's not okay to object to a veto on the grounds that the change in question has already been committed, or the action taken (unless it's something irrevocable, like putting out a press release). On the other hand, a veto that arrives weeks or months late isn't likely to be taken very seriously, nor should it be.

Writing It All Down

At some point, the number of conventions and agreements floating around in your project may become so great that you need to record it somewhere. In order to give such a document legitimacy, make it clear that it is based on mailing list discussions and on agreements already in effect. As you compose it, link to the relevant threads in the mailing list archives, and whenever there's a point you're not sure about, ask again. The document should not contain any surprises: remember, it is not the source of the agreements, it is merely a description of them. Of course, if it is successful, people will start citing it as a source of authority in itself, but that just means it reflects the overall will of the group accurately.

Linking To Emails

When you link to an email thread in the archives, it's a good practice to give not only the thread's URL, but the subject, sender name, and date of the first message in the thread (or at least of some message in the thread). The reason is that archives sometimes move — surprisingly often, actually — and the URL alone will usually not contain enough information to find the message or thread in its new location.

The same advice could apply to bug tickets too, but in practice bug trackers move less often than mail archives do, and when a bug tracker moves the project usually manages to either preserve the ticket numbers or make a mapping between old and new ticket numbers, so that old references can be resolved with a little extra effort. For various technical reasons, this is harder to do with emails and especially with threads, so the better solution is just for references to include enough information to do a search in the new archive if necessary. See also the section called “Conspicuous Use of Archives”.

This is the document alluded to in the section called “Developer Guidelines” in Chapter 2, *Getting Started*. Naturally, when the project is very young, you will have to lay down guidelines without the benefit of a long project history to draw on. But as the development community matures, you can adjust the language to reflect the way things actually turn out.

Don't try to be comprehensive. No document can capture everything people need to know about participating in a project. Many of the conventions a project evolves remain forever unspoken, never mentioned explicitly, yet adhered to by all. Other things are simply too obvious to be mentioned, and would only distract from important but non-obvious material. For example, there's no point writing guidelines like "Be polite and respectful to others on the mailing lists, and don't start flame wars," or "Write clean, readable bug-free code." Of course these things are desirable, but since there's no conceivable universe in which they might *not* be desirable, they are not worth mentioning. If people are being rude on the mailing list, or writing buggy code, they're not going to stop just because the project guidelines said to. Such situations need to be dealt with as they arise, not by blanket admonitions to be good. On the other hand, if the project has specific guidelines about *how* to write good code, such as rules about documenting every API in a certain format, then those guidelines should be written down as completely as possible.

A good way to determine what to include is to base the document on the questions that newcomers ask most often, and on the complaints experienced developers make most often. This doesn't necessarily mean it should turn into a FAQ sheet—it probably needs a more coherent narrative structure than FAQs can offer. But it should follow the same reality-based principle of addressing the issues that actually arise, rather than those you anticipate might arise.

If the project is a benevolent dictatorship, or has officers endowed with special powers (president, chair, whatever), then the document is also a good opportunity to codify succession procedures. Sometimes this can be as simple as naming specific people as replacements in case the BD suddenly leaves the project for any reason. Generally, if there is a BD, only the BD can get away with naming a successor. If there are elected officers, then the nomination and election procedure that was used to choose them in the first place should be described in the document. If there was no procedure originally, then get consensus on a procedure on the mailing lists *before* writing about it. People can sometimes be touchy about hierarchical structures, so the subject needs to be approached with sensitivity.

Perhaps the most important thing is to make it clear that the rules can be reconsidered. If the conventions described in the document start to hamper the project, remind everyone that it is supposed to be a living reflection of the group's intentions, not a source of frustration and blockage. If someone makes a habit of inappropriately asking for rules to be reconsidered every time the rules get in her way, you don't always need to debate it with her—sometimes silence is the best tactic. If other people agree with

the complaints, they'll chime in, and it will be obvious that something needs to change. If no one else agrees, then the person won't get much response, and the rules will stay as they are.

Three good examples of project guidelines are the LibreOffice Development guide at wiki.documentfoundation.org/Development [https://wiki.documentfoundation.org/Development], theSubversion Community Guide, at subversion.apache.org/docs/community-guide/ [https://subversion.apache.org/docs/community-guide/], and the Apache Software Foundation governance documents, at apache.org/foundation/how-it-works.html [https://www.apache.org/foundation/how-it-works.html] and apache.org/foundation/voting.html [https://www.apache.org/foundation/voting.html]. The ASF is really a collection of software projects, legally organized as a nonprofit corporation, so its documents tend to describe governance procedures more than development conventions. They're still worth reading, though, because they represent the accumulated experience of a lot of open source projects.

Joining or Creating a Non-Profit Organization

Successful open source projects often get to a point where they feel the need for some sort of formal existence as a legal entity — to be able to accept donations (see Chapter 5, *Organizations, Money, and Business* for discussion of how to handle incoming funding), to purchase and maintain infrastructure for the project's benefit, to organize conferences and developer meetups, to enforce trademarks, etc.

There may be a few exceptional circumstances where forming your own organization from scratch would be the right solution, but for most projects it is much better to join an existing organization. There are umbrella organizations whose purpose is to provide a legal home for open source projects. Working with multiple projects gives these organizations economies of scale and broad experience — any of them would almost certainly do a better job of providing services to your project than your project could manage if it started its own organization.

Here are some well-known and reputable umbrella organizations:

- Software Freedom Conservancy (sfconservancy.org [https://sfconservancy.org/])[4]
- Apache Software Foundation apache.org [https://apache.org/]
- Eclipse Foundation eclipse.org [https://eclipse.org/]
- OuterCurve Foundation outercurve.org [http://outercurve.org/]
- Software in the Public Interest spi-inc.org [http://spi-inc.org/]
- Linux Foundation collabprojects.linuxfoundation.org [http://collabprojects.linuxfoundation.org/]

These are all based in the United States, but there are similar umbrella organizations outside the U.S. — I just didn't know them well enough to make recommendations. If you're a U.S. reader, remember that the distinctions the U.S. tax code makes between different types of non-profit corporations, such as 501(c)(3) tax-exempt organizations vs 501(c)(6) trade associations, may not be meaningful to members of your project outside the U.S., and that the tax benefits available to donors under 501(c)(3) won't apply to non-U.S. donors anyway.

If your project joins or creats a non-profit organization, make clear the separation between the legal infrastructure and the day-to-day running of the project. The organization is there to handle things the developers don't want to handle, not to interfere with the things the developers *do* want to handle and are already competent to handle. Even if the non-profit becomes the official owner of the project's copy-

[4] I think the Software Freedom Conservancy is a good choice for most projects, which is why I listed it first. But I should add the disclaimer that I joined their Evaluation Committee, a volunteer committee that evaluates projects applying to become members of the Conservancy, while revising this book for its 2nd edition. The recommendation of the Conservancy was already in the in-progress text before I joined the committee.

rights, trademarks, and other assets, that shouldn't change the way decisions are made about technical questions, project direction, etc. One of the reasons to join one of the existing organizations is that they already have experience with this distinction, and know how to fairly read the will of the project even when there is controversy or strong disagreement. They also serve as a neutral place for resolving disagreements about how to allocate the project's money or other resources. More than one of the organizations above has had to play "project psychotherapist" on occasion, and their ability to do so should be considered an advantage even by a healthy and smoothly functioning project.

Chapter 5. Organizations, Money, and Business

This chapter examines how to use money constructively in a free software environment, and some of the adjustments your organization might want to consider as it gets involved in free software projects. This chapter is aimed partly at developers who are paid to work on open source projects, but even more at their managers and even at the executives making strategic decisions. When an organization makes an investment in open source, people at all levels have to understand not just how best to structure that investment, but the effects that long-term open source engagement will have on the organization itself. Open source can be transformative — at least when done right.

This chapter is *not* primarily about how to find funding sources for your open source project, though I hope it will usefully inform that topic. There are many different ways open source projects are funded [1], just as there are many ways all human endeavors are funded. While open source is incompatible with one particular business model — monopoly-controlled royalty streams based on per-copy sales — it is compatible with all the others, and indeed is better suited to some of them than proprietary software is.

The Economics of Open Source

People are still sometimes surprised to learn that most free software is written by paid developers, not by volunteers. But the economics that drive open source are actually quite straightforward: a company often needs a particular piece of software to be maintained and developed, and yet does not need a monopoly on that software. Indeed, it would often be disadvantageous to have a monopoly, because then the entire burden of maintenance would fall on that one company, instead of being shared with others who have the same needs. For example, most companies have web sites and therefore need a web server, but almost no companies need exclusive control over the development of their web server, or intend to sell copies of it on a proprietary basis. The same is true of office software suites, operating system kernels, network connectivity tools, educational programs, etc — just as historically it has been true of electric grids, roads, sewer systems, and other goods that everyone needs but no one needs to own. Just as we expect road workers to be paid, we should expect software developers to be paid as well.

Even in the early days of free software, when the proportion of truly unpaid volunteers was probably higher than it is now, there were already developers who were paid for their work. There was also a lot of informal subsidy, as there continues to be today. When a system administrator writes a network analysis tool to help her do her job, then posts it online and gets bug fixes and feature contributions from other system administrators, what's happened is that an unofficial consortium has been formed. The consortium's funding comes from the sysadmins' salaries; its office space and network bandwidth are donated, albeit unknowingly, by the organizations those people work for. Those organizations also benefit from the investment, of course, although they may or may not be institutionally aware of it.

Today such efforts are often more formalized. Corporations have become conscious of the benefits of open source software, and now involve themselves intentionally in its development. Developers too have come to expect that really important projects will attract funding in one way or another. The key question is how the hierarchical command structures of corporations and the semi-decentralized, non-coercive communities of free software projects can work productively with each other, and how they can more or less agree on what "productively" means.

Financial backing is generally welcomed by open source development communities. Having paid developers mean that bug reports are more likely to be listened to, that needed work is more likely to get

[1] See en.wikipedia.org/wiki/Business_models_for_open-source_software [https://en.wikipedia.org/wiki/Business_models_for_open-source_software] for an incomplete list.

done, and that the project will be less vulnerable to the Forces of Chaos (e.g., a key founding developer suddenly losing interest) lurking at the edges of every collaborative endeavor. One key dynamic is that credibility is contagious, to a point. When, for example, Google backs an open source project, people assume the project will have the chance to succeed or fail on its long-term merits, while receiving adequate support in its early stages; other participants' resultant willingness to devote effort to it can then make this a self-fulfilling prophecy.

However, funding can also bring a perception of control. If not handled carefully, money can divide a project into in-group and out-group developers. If developers who aren't officially paid to work on the project get the feeling that design decisions or feature additions are simply available to the highest bidder, they'll head off to a project that seems more like a meritocracy and less like unpaid labor for someone else's benefit. They may never complain overtly on the mailing lists. Instead, there will simply be less and less noise from external sources, as the "out" developers gradually stop trying to be taken seriously. The buzz of small-scale activity will continue, in the form of bug reports and occasional small fixes. But there will be fewer and fewer large code contributions from unexpected sources, fewer unexpected opinions voiced in design discussions, and so on. People sense what's expected of them, and live up (or down) to those expectations.

Although money needs to be used carefully, that doesn't mean it can't buy influence. It most certainly can. The trick is that it doesn't buy influence directly — instead, it buys development credibility, which is convertible to influence through the project's decision-making processes. In a straightforward commercial transaction, you trade money for what you want. If you need a feature added, you sign a contract, pay for it, and (if all goes well) the work gets done and the feature eventually lands in the product. In an open source project, it's not so simple. You may sign a contract with some developers, but they'd be fooling themselves—and you—if they guaranteed that the work you paid for would be accepted by the development community simply because you paid for it. The work can only be accepted on its own merits and on how it fits into the community's vision for the software (see the section called “Contracting”). You may have some say in that vision, but you won't be the only voice.

So money can't purchase influence directly, but it can purchase things that *lead to* influence. The most obvious example is programmers. If you hire good programmers, and they stick around long enough to get experience with the software and credibility in the community, then they can influence the project by the same means as any other member. They will have a vote, or if there are many of them, they will have a voting block[2]. If they are respected in the project, they will have influence beyond just their votes. There is no need for paid developers to disguise their motives, either. After all, everyone who wants a change made to the software wants it for a reason. Your company's reasons are no less legitimate than anyone else's. It's just that the weight given to your company's goals will be determined by its representatives' status in the project, not by your company's size, budget, or business plan.[3]

Types of Corporate Involvement

There are many different reasons open source projects get corporate support. This list is just a high-level survey. The items in it aren't mutually exclusive; often a project's financial backing will result from several, or even all, of these motivations:

[2]Even though actual votes may be rare, as noted in the section called “Consensus-based Democracy”, the *possibility* of a vote has great implicit power, so membership in the electorate is still important even if no vote is ever held.

[3]When companies need to guarantee that certain features and bug fixes land in a specified amount of time, they accomplish this by keeping their own copy (which may be partially or wholly public) of the project, and merging it from time to time with a separate public copy that has its own governance. Google's Android operating system is a classic example: Google maintains its own copy of Android, which it governs as it pleases, and from time to time merges changes between that copy and the Android Open Source Project [https://en.wikipedia.org/wiki/Android_%28operating_system%29#Open-source_community]. Essentially, Google is on a very long copy-modify-merge loop with respect to the open source project, and vice versa. It is in neither side's interests to permanently diverge from the other.

Sharing the burden

Separate organizations with related software needs often find themselves duplicating effort, either by redundantly writing similar code in-house, or by purchasing similar products from proprietary vendors. As the inefficiency becomes clear to the different parties, they may pool their resources — often gradually, without at first realizing the overall trajectory of the process — and create (or join) an open source project tailored to their needs. The advantages are obvious: the costs of development are divided, but the benefits accrue to all. Although this scenario might seem most intuitive for nonprofits, in practice it often happens even among for-profit competitors.

Augmenting services

When a company sells services which depend on, or are made more attractive by, particular open source programs, it is naturally in that company's interests to ensure those programs are actively maintained.

Creating an ecosystem

For investors who like to think big, the right open source effort can create a new ecosystem — one in which those investors are more likely to flourish. A good example of this kind of investment, as of this writing in 2014, is the Meteor.com [https://www.meteor.com/] project.

Supporting hardware sales

The value of computers and computer components is directly related to the amount of software available for them. Hardware vendors—not just whole-machine vendors, but also makers of peripheral devices and microchips—have found that having high-quality free software to run on their hardware is important to customers.

Undermining a competitor

Sometimes companies support a particular open source project as a means of undermining a competitor's product, which may or may not be open source itself. Eating away at a competitor's market share is usually not the sole reason for getting involved with an open source project, but it can be a factor.

Marketing

Having your company associated with a popular open source application can be good brand management, not just in the eyes of customers but in the eyes of potential employees.

Proprietary relicensing

Proprietary relicensing is the practice of offering software under a traditional proprietary license for customers who want to resell it as part of a proprietary application of their own, and simultaneously under a free license for those willing to use it under open source terms (see the section called “Proprietary Relicensing” in Chapter 9, *Legal Matters: Licenses, Copyrights, Trademarks and Patents*). If the open source developer community is active, the software gets the benefits of wide-area debugging and development, yet the company still gets a royalty stream to support some full-time programmers.

Proprietary relicensing is controversial because it is not a "pure open source play" (as we say in business-speak), but rather yokes funding for open source development to a monopoly-based revenue stream. Whether this is a problem for you depends on where you fall on the "open source is just a way of software development" to "open source is a way of life" spectrum. The presence of

revenue from a proprietary version does not *necessarily* mean that the free software version is worse off, and some very well-known and widely-used free software has had corresponding proprietary versions (MySQL [https://en.wikipedia.org/wiki/MySQL] is probably the most famous example). However, some developers dislike the thought that their contributions may end up in the proprietary version. Also, the mere presence of the proprietary version suggests the possibility that some of the best salaried developers' attention is going to the proprietary code, not the open source code. This tends to undermine other developers' faith in the open source project, which in turn makes it difficult to develop a truly flourishing ecosystem around the open source version.

None of is meant to persuade you not to do proprietary relicensing. You should just be aware that this strategy is unlike the other business approaches I've listed here, that it requires more care and sophistication to manage successfully, and that it is usually incompatible with the presence of a committed and involved ecosystem of developers from outside your organization, particularly developers who might have their own commercial motivations.

Donations

A widely-used project can sometimes get significant contributions, from both individuals and organizations, just by soliciting donations, or by doing an organized crowdfunding campaign. See the section called “Crowdfunding and Bounties” for a detailed discussion of this approach.

A funder's business model is not the only factor in how that funder relates to an open source community. The historical relationship between the two also matters: did the company start the project, or is it joining an existing development effort? In both cases, the funder will have to earn credibility, but, not surprisingly, there's a bit more earning to be done in the latter case. The organization needs to have clear goals with respect to the project. Is the company trying to keep a position of leadership, or simply trying to be one voice in the community, to guide but not necessarily govern the project's direction? Or does it just want to have a couple of committers around, able to fix customers' bugs and get the changes into the public distribution without any fuss?

Keep these questions in mind as you read the guidelines that follow. They are meant to apply to any sort of organizational involvement in a free software project, but every project is a human environment, and therefore no two are exactly alike. To some degree, you will always have to play by ear, but following these principles will increase the likelihood of things turning out the way you want.

Governments and Open Source

Since the first edition of this book came out in 2005, I've worked with various U.S. government agencies, at the federal, state, and municipal levels, to help them develop and release open source software. I've also been lucky enough to observe, and in a few cases work with, some government agencies outside the U.S. These experiences have convinced me of one thing: *government is different*. If you work at a government agency and the material in this book so far has made you shake your head and think "Sure, but it'll never work here", you have my sympathy — I know exactly what you mean. Governments differ from individuals and from private-sector organizations in some fundamental ways:

- Governments often aren't trying to retain technical expertise in-house. That's what contractors are for, after all.

- Governments have labyrinthine and in certain ways inflexible procurement and employment policies. These policies can make it difficult for a government agency to be nimbly responsive in an open source development community.

- Government agencies tend to be unusually risk-averse. Somewhere at the top there's an elected official who, reasonably, sees an open source project as just one more surface area for opponents to at-

tack. After all, when development happens in public, the inevitable false starts and wrong turns are also public; if development were internal, no one else would know about it when those things happen.

- Government officials hunger for well-timed and well-controlled publicity events. This has certain benefits, but it can sometimes come at the expense of overall project health. This need for publicity is the complement of being risk-averse: elected officials and those who work for them understand that most people aren't paying much attention most of the time — therefore, those who work in government want to ensure that in the few moments when people *are* paying attention, they see something good. This is understandable, but it can cause them to delay certain actions — or, in some cases, do them too soon — based on external publicity implications rather than on what's best for the project technically and socially.

There are good reasons for all of these things; they've been true for decades if not centuries, and they're not going to change. So if you're a government agency and you want to start a successful open source project, certain adjustments will be necessary to compensate for the structural idiosyncracies mentioned above. Much of that advice is also applicable to non-governmental organizations, and is already present elsewhere in this chapter, so what I'll do here is simply list the sections that I think are *most* important for a government agency:

- Update Your RFI, RFP and Contract Language
- IV&V: Use Third-Party Review Throughout Development
- Don't Surprise Your Lawyers
- Open Source and Freedom from Vendor Lock-In
- Dispel Myths Within Your Organization
- Don't Let Publicity Events Drive Project Schedule
- The Key Role of Middle Management

In addition to the above sections, there are many excellent online resources for doing open source in government — I won't even try to include a complete list, as there is too much and it changes too quickly. Here are a few sites that are likely to remain good starting points for some time to come, especially for government agencies in the United States and in countries with procurement and civil service systems similar to those of the U.S. 18f.gsa.gov [https://18f.gsa.gov/] is a digital services agency within the United States federal government, created in 2014 to bring modern software development practices to government work. 18F serves as a technology consultancy to other agencies, and builds its deliverables out in the open as open source software. Along the way, 18F has generated useul guidelines and observations that anyone trying to run an open source software project within government can benefit from. Mil-OSS.org [http://mil-oss.org/] is an association of civilian and military open source software and hardware developers in the U.S, but the advice and resources they've put together broadly applicable to open source in government generally, not just to military projects. The home site of Dr. David A. Wheeler, dwheeler.com [http://www.dwheeler.com/], is a fantastic trove that includes, among many other open-source-related things, tons of information about how to use U.S. government procurement regulations to support open source development. If you are advocating for open source development within a government agency, ben.balter.com/2015/11/23/why-open-source [http://ben.balter.com/2015/11/23/why-open-source/] is a terrific post to mine for arguments, and many of Ben Balter's other writings are worth looking at too.

Finally, there is one issue in particular that I have encountered over and over again in government-initiated open source projects. It is so common, and so potentially damaging to a project, that I have given it its own subsection below.

Being Open Source From Day One is Especially Important for Government Projects

In the section called “Be Open From Day One” in Chapter 2, *Getting Started*, I explained why it's best for an open source project to be run in the open from the very beginning. That advice, particularly the section called “Waiting Just Creates an Exposure Event”, is if anything even more true for government code.

Government projects have greater potential to be harmed by a needless exposure event than private-sector projects have. Elected officials and those who work for them are understandably sensitive to negative public comments. Thus even for the most conscientious team, a worrying cloud of uncertainty will surround everything by the time you're ready to open up hitherto closed code. How can you ever know you've got it all cleaned up? You do your best, but you can never be totally sure some hawk-eyed hacker out there won't spot something embarrassing after the release. The team worries, and worry is an energy drain: it causes them to spend time chasing down ghosts, and at the same time can cause them to unconsciously avoid steps that might risk revealing real problems.

This concern doesn't only apply to government software, of course. But in the private sector, businesses sometimes have competitive reasons to stay behind the curtain until their first release, even if they intend for the project to be open source in the long run. Government projects should not have that motivation to start out closed, at least in theory, and they have even more to lose.

Hire for the Long Term

If you're managing programmers on an open source project, keep them there long enough that they acquire both technical and political expertise—a couple of years, at a minimum. Of course, no project, whether open or closed-source, benefits from swapping programmers in and out too often. The need for a newcomer to learn the ropes each time would be a deterrent in any environment. But the penalty is even stronger in open source projects, because outgoing developers take with them not only their knowledge of the code, but also their status in the community and the human relationships they have made there.

The credibility a developer has accumulated cannot be transferred. To pick the most obvious example, an incoming developer can't inherit commit access from an outgoing one (see the section called “Money Can't Buy You Love” later in this chapter), so if the new developer doesn't already have commit access, he will have to submit patches until he does. But commit access is only the most easily quantifiable manifestation of lost influence. A long-time developer also knows all the old arguments that have been hashed and rehashed on the discussion lists. A new developer, having no memory of those conversations, may try to raise the topics again, leading to a loss of credibility for your organization; the others might wonder "Can't they remember anything?" A new developer will also have no political feel for the project's personalities, and will not be able to influence development directions as quickly or as smoothly as one who's been around a long time.

Train newcomers through a program of supervised engagement. The new developer should be in direct contact with the public development community from the very first day, starting off with bug fixes and cleanup tasks, so he can learn the codebase and acquire a reputation in the community, yet not spark any long and involved design discussions. All the while, one or more experienced developers should be available for questioning, and should be reading every post the newcomer makes to the development lists, even if they're in threads that the experienced developers normally wouldn't pay attention to. This will help the group spot potential rocks before the newcomer runs aground. Private, behind-the-scenes encouragement and pointers can also help a lot, especially if the newcomer is not accustomed to massively parallel peer review of his code.

Case study

At CollabNet, when we hired a new developer to work on Subversion, we would sit down together and pick some open bugs for the new person to cut his teeth on. We'd discuss the technical outlines of the solutions, and then assign at least one experienced developer to (publicly) review the patches that the new developer would (also publicly) post. We typically didn't even look at the patch before the main development list saw it, although we could if there were some reason to. The important thing is that the new developer goes through the process of public review, learning the codebase while simultaneously becoming accustomed to receiving critiques from complete strangers. But we also tried to coordinate the timing so that our own review came immediately after the posting of the patch. That way the first review the list sees is ours, which can help set the tone for the others' reviews. It also contributes to the idea that this new person is to be taken seriously: if others see that we're putting in the time to give detailed reviews, with thorough explanations and references into the archives where appropriate, they'll appreciate that a form of training is going on, and that it probably signifies a long-term investment. This can make them more positively disposed toward the developer, to the degree of spending a little extra time answering questions and reviewing patches themselves.

Appear as Many, Not as One

Your developers should strive to appear in the project's public forums as individual participants, rather than as a monolithic corporate presence. This is not because there is some negative connotation inherent in monolithic corporate presences (well, perhaps there is, but that's not what this book is about). Rather, it's because individuals are the only sort of entity open source projects are structurally equipped to deal with. An individual contributor can have discussions, submit patches, acquire credibility, vote, and so forth. A company cannot.

Furthermore, by behaving in a decentralized manner, you avoid stimulating centralization of opposition. Let your developers disagree with each other on the mailing lists. Encourage them to review each other's code as often, and as publicly, as they would anyone else's. Discourage them from always voting as a bloc, because if they do, others may start to feel that, just on general principles, there should be an organized effort to keep them in check.

There's a difference between actually being decentralized and simply striving to appear that way. Under certain circumstances, having your developers behave in concert can be quite useful, and they should be prepared to coordinate behind the scenes when necessary. For example, when making a proposal, having several people chime in with agreement early on can help it along, by giving the impression of a growing consensus. Others will feel that the proposal has momentum, and that if they were to object, they'd be stopping that momentum. Thus, people will object only if they have a good reason to do so. There's nothing wrong with orchestrating agreement like this, as long as objections are still taken seriously. The public manifestations of a private agreement are no less sincere for having been coordinated beforehand, and are not harmful as long as they are not used to prejudicially snuff out opposing arguments. Their purpose is merely to inhibit the sort of people who like to object just to stay in shape; see the section called “The Smaller the Topic, the Longer the Debate” in Chapter 6, *Communications* for more about them.

Be Open About Your Motivations

Be as open about your organization's goals as you can without compromising business secrets. If you want the project to acquire a certain feature because, say, your customers have been clamoring for it, just say so outright on the mailing lists. If the customers wish to remain anonymous, as is sometimes the case, then at least ask them if they can be used as unnamed examples. The more the public development community knows about *why* you want what you want, the more comfortable they'll be with whatever you're proposing.

This runs counter to the instinct—so easy to acquire, so hard to shake off—that knowledge is power, and that the more others know about your goals, the more control they have over you. But that instinct would be wrong here. By publicly advocating the feature (or bugfix, or whatever it is), you have *already* laid your cards on the table. The only question now is whether you will succeed in guiding the community to share your goal. If you merely state that you want it, but can't provide concrete examples of why, your argument is weak, and people will start to suspect a hidden agenda. But if you give just a few real-world scenarios showing why the proposed feature is important, that can have a dramatic effect on the debate.

To see why this is so, consider the alternative. Too frequently, debates about new features or new directions are long and tiresome. The arguments people advance often reduce to "I personally want X," or the ever-popular "In my years of experience as a software designer, X is extremely important to users" or "...is a useless frill that will please no one." Predictably, the absence of real-world usage data neither shortens nor tempers such debates, but instead allows them to drift farther and farther from any mooring in actual user experience. Without some countervailing force, the end result is likely to be determined by whoever was the most articulate, or the most persistent, or the most senior.

As an organization with plentiful customer data available, you have the opportunity to provide just such a countervailing force. You can be a conduit for information that might otherwise have no means of reaching the development community. The fact that the information supports your desires is nothing to be embarrassed about. Most developers don't individually have very broad experience with how the software they write is used. Each developer uses the software in her own idiosyncratic way; as far as other usage patterns go, she's relying on intuition and guesswork, and deep down, she knows this. By providing credible data about a significant number of users, you are giving the public development community something akin to oxygen. As long as you present it right, they will welcome it enthusiastically, and it will propel things in the direction you want to go.

The key, of course, is presenting it right. It will never do to insist that simply because you deal with a large number of users, and because they need (or think they need) a given feature, therefore your solution ought to be implemented. Instead, you should focus your initial posts on the problem, rather than on one particular solution. Describe in great detail the experiences your customers are encountering, offer as much analysis as you have available, and as many reasonable solutions as you can think of. When people start speculating about the effectiveness of various solutions, you can continue to draw on your data to support or refute what they say. You may have one particular solution in mind all along, but don't single it out for special consideration at first. This is not deception, it is simply standard "honest broker" behavior. After all, your true goal is to solve the problem; a solution is merely a means to that end. If the solution you prefer really is superior, other developers will recognize that on their own eventually—and then they will get behind it of their own free will, which is much better than you browbeating them into implementing it. (There is also the possibility that they will think of a better solution.)

This is not to say that you can't ever come out in favor of a specific solution. But you must have the patience to see the analysis you've already done internally repeated on the public development lists. Don't post saying "Yes, we've been over all that here, but it doesn't work for reasons A, B, and C. When you get right down to it, the only way to solve this is Q." The problem is not so much that it sounds arrogant as that it gives the impression that you have *already* devoted some unknown (but, people will presume, large) amount of analytical resources to the problem, behind closed doors. It makes it seem as though efforts have been going on, and perhaps decisions made, that the public is not privy to—and that is a recipe for resentment.

Naturally, *you* know how much effort you've devoted to the problem internally, and that knowledge is, in a way, a disadvantage. It puts your developers in a slightly different mental space than everyone else on the mailing lists, reducing their ability to see things from the point of view of those who haven't yet thought about the problem as much. The earlier you can get everyone else thinking about things in the same terms as you do, the smaller this distancing effect will be. This logic applies not only to individual technical situations, but to the broader mandate of making your goals as clear as you can. The unknown is always more destabilizing than the known. If people understand why you want what you want, they'll

feel comfortable talking to you even when they disagree. If they can't figure out what makes you tick, they'll assume the worst, at least some of the time.

You won't be able to publicize everything, of course, and people won't expect you to. All organizations have secrets; perhaps for-profits have more of them, but nonprofits have them too. If you must advocate a certain course, but can't reveal anything about why, then simply offer the best arguments you can under that handicap, and accept the fact that you may not have as much influence as you want in the discussion. This is one of the compromises you make in order to have a development community not on your payroll.

Money Can't Buy You Love

If you're a paid developer on a project, then set guidelines early on about what the money can and cannot buy. This does not mean you need to post twice a day to the mailing lists reiterating your noble and incorruptible nature. It merely means that you should be on the lookout for opportunities to defuse the tensions that *could* be created by money. You don't need to start out assuming that the tensions are there; you do need to demonstrate an awareness that they have the potential to arise.

A perfect example of this came up early in the Subversion project. Subversion was started in 2000 by CollabNet [http://www.collab.net/], which has been the project's primary funder since its inception, paying the salaries of several developers (disclaimer: I was one of them for six years). Soon after the project began, we hired another developer, Mike Pilato, to join the effort. By then, coding had already started. Although Subversion was still very much in the early stages, it already had a development community with a set of basic ground rules.

Mike's arrival raised an interesting question. Subversion already had a policy about how a new developer gets commit access. First, he submits some patches to the development mailing list. After enough patches have gone by for the other committers to see that the new contributor knows what he's doing, someone proposes that he just commit directly (that proposal is private, as described in the section called “Committers”). Assuming the committers agree, one of them mails the new developer and offers him direct commit access to the project's repository.

CollabNet had hired Mike specifically to work on Subversion. Among those who already knew him, there was no doubt about his coding skills or his readiness to work on the project. Furthermore, the other developers had a very good relationship with the CollabNet employees, and most likely would not have objected if we'd just given Mike commit access the day he was hired. But we knew we'd be setting a precedent. If we granted Mike commit access by fiat, we'd be saying that CollabNet had the right to ignore project guidelines, simply because it was the primary funder. While the damage from this would not necessarily be immediately apparent, it would gradually result in the non-salaried developers feeling disenfranchised. Other people have to earn their commit access—CollabNet just buys it.

So Mike agreed to start out his employment at CollabNet like any other new developer, without commit access. He sent patches to the public mailing list, where they could be, and were, reviewed by everyone. We also said on the list that we were doing things this way deliberately, so there could be no missing the point. After a couple of weeks of solid activity by Mike, someone (I can't remember if it was a CollabNet developer or not) proposed him for commit access, and he was accepted, as we knew he would be.

That kind of consistency gets you a credibility that money could never buy. And credibility is a valuable currency to have in technical discussions: it's immunization against having one's motives questioned later. In the heat of argument, people will sometimes look for non-technical ways to win the battle. The project's primary funder, because of its deep involvement and obvious concern over the directions the project takes, presents a wider target than most. By being scrupulous to observe all project guidelines right from the start, the funder makes itself the same size as everyone else.

(See also Danese Cooper's blog post, preserved in the Internet Archive's Wayback Machine at web.archive.org/web/20050227033105/http://blogs.sun.com/roller/page/DaneseCooper/20040916 [https://web.archive.org/web/20050227033105/http://blogs.sun.com/roller/page/DaneseCooper/20040916], for a similar story about commit access. Cooper was then Sun Microsystem's "Open Source Diva"—I believe that was her official title—and in the blog entry, she describes how the Tomcat development community got Sun to hold its own developers to the same commit-access standards as the non-Sun developers.)

The need for the funders to play by the same rules as everyone else means that the Benevolent Dictatorship governance model (see the section called “Benevolent Dictators” in Chapter 4, *Social and Political Infrastructure*) is slightly harder to pull off in the presence of funding, particularly if the dictator works for the primary funder. Since a dictatorship has few rules, it is hard for the funder to prove that it's abiding by community standards, even when it is. It's certainly not impossible; it just requires a project leader who is able to see things from the point of view of the outside developers, as well as that of the funder, and act accordingly. Even then, it's probably a good idea to have a proposal for non-dictatorial governance sitting in your back pocket, ready to be brought out the moment there are any indications of widespread dissatisfaction in the community.

Contracting

Contracted work needs to be handled carefully in free software projects. Ideally, if you hire a contractor you want her work to be accepted by the community and folded into the public distribution. In theory, it wouldn't matter who the contractor is, as long as her work is good and meets the project's guidelines. Theory and practice can sometimes match, too: a complete stranger who shows up with a good patch *will* generally be able to get it into the software. The trouble is, it's very hard to produce an acceptable patch for a non-trivial enhancement or new feature while truly being a complete stranger; one must first discuss it with the rest of the project. The duration of that discussion cannot be precisely predicted. If the contractor is paid by the hour, you may end up paying more than you expected; if she is paid a flat sum, she may end up doing more work than she can afford.

There are two ways around this. The preferred way is to make an educated guess about the length of the discussion process, based on past experience, add in some padding for error, and base the contract on that. It also helps to divide the problem into as many small, independent chunks as possible, to increase the predictability of each chunk. The other way is to contract solely for delivery of a patch, and treat the patch's acceptance into the public project as a separate matter. Then it becomes much easier to write the contract, but you're stuck with the burden of maintaining a private patch for either as long as you depend on the software or for as long as it takes you to get that patch into the upstream codebase.

Even with the preferred way, the contract itself cannot require that the patch be accepted by the upstream project, because that would involve selling something that's not for sale. (What if the rest of the project unexpectedly decides not to support the feature?) However, the contract can require a *bona fide* effort to get the change accepted by the community, and that it be committed to the repository if the community agrees with it. For example, if the project has written standards (e.g., about coding conventions, documentation, writing regression tests, submitting patches, etc), the contract can reference those standards and specify that the contracted work must meet them. In practice, this usually works out the way everyone hopes.

The best tactic for successful contracting is to hire one of the project's developers—preferably a committer—as the contractor. This may seem like a form of purchasing influence, and, well, it is. But it's not as corrupt as it might seem. A developer's influence in the project is due mainly to the quality of her code and to her interactions with other developers. The fact that she has a contract to get certain things done doesn't raise her status in any way, and doesn't lower it either, though it may make people scrutinize her more carefully. Most developers would not risk their long-term position in the project by backing an inappropriate or widely disliked new feature. In fact, part of what you get, or should get, when you hire such a contractor is advice about what sorts of changes are likely to be accepted by the community. You

also get a slight shift in the project's priorities. Because prioritization is just a matter of who has time to work on what, when you pay for someone's time, you cause their work to move up in the priority queue a bit. This is a well-understood fact of life among experienced open source developers, and at least some of them will devote attention to the contractor's work simply because it looks like it's going to *get done*, so they want to help it get done right. Perhaps they won't write any of the code, but they'll still discuss the design and review the code, both of which can be very useful. For all these reasons, the contractor is best drawn from the ranks of those already involved with the project.

This immediately raises two questions: Should contracts ever be private? And when they're not, should you worry about creating tensions in the community by the fact that you've contracted with some developers and not others?

It's best to be open about contracts when you can. Otherwise, the contractor's behavior may seem strange to others in the community—perhaps she's suddenly giving inexplicably high priority to features she's never shown interest in the past. When people ask her why she wants them now, how can she answer convincingly if she can't talk about the fact that she's been contracted to write them?

At the same time, neither you nor the contractor should act as though others should treat your arrangement as a big deal. Too often I've seen contractors waltz onto a development list with the attitude that their posts should be taken more seriously simply because they're being paid. That kind of attitude signals to the rest of the project that the contractor regards the fact of the contract—as opposed to the code *resulting* from the contract—to be the important thing. But from the other developers' point of view, only the code matters. At all times, the focus of attention should be kept on technical issues, not on the details of who is paying whom. For example, one of the developers in the Subversion community handles contracting in a particularly graceful way. While discussing his code changes in IRC, he'll mention as an aside (often in a private remark, an IRC *privmsg*, to one of the other committers) that he's being paid for his work on this particular bug or feature. But he also consistently gives the impression that he'd want to be working on that change anyway, and that he's happy the money is making it possible for him to do that. He may or may not reveal his customer's identity, but in any case he doesn't dwell on the contract. His remarks about it are just an ornament to an otherwise technical discussion about how to get something done.

That example shows another reason why it's good to be open about contracts. There may be multiple organizations sponsoring contracts on a given open source project, and if each knows what the others are trying to do, they may be able to pool their resources. In the above case, the project's largest funder (CollabNet) was not involved with these piecework contracts, but knowing that someone else was sponsoring certain bug fixes allowed CollabNet to redirect its resources to other bugs, resulting in greater efficiency for the project as a whole.

Will other developers resent that some are paid for working on the project? In general, no, particularly when those who are paid are established, well-respected members of the community anyway. No one expects contract work to be distributed equally among all the committers. People understand the importance of long-term relationships: the uncertainties involved in contracting are such that once you find someone you can work reliably with, you would be reluctant to switch to a different person just for the sake of evenhandedness. Think of it this way: the first time you hire, there will be no complaints, because clearly you had to pick *someone*—it's not your fault you can't hire everyone. Later, when you hire the same person a second time, that's just common sense: you already know her, the last time was successful, so why take unnecessary risks? Thus, it's perfectly natural to have a few go-to people in the community, instead of spreading the work around evenly.

Review and Acceptance of Changes

The project's community will always be important to the long-term success of contract work. Their involvement in the design and review process for sizeable changes cannot be an afterthought; It must be

considered part of the work, and fully embraced by the contractor. Don't think of community scrutiny as an obstacle to be overcome—think of it as a free design board and QA department. It is a benefit to be aggressively pursued, not merely endured.

Case study: the CVS password-authentication protocol

In 1995, I was one half of a partnership that provided support and enhancements for CVS (the Concurrent Versions System; see nongnu.org/cvs [http://nongnu.org/cvs]). My partner Jim Blandy and I were, informally, the maintainers of CVS by that point. But we'd never thought carefully about how we ought to relate to the existing mostly part-time and volunteer CVS development community. We just assumed that they'd send in patches, and we'd apply them, and that was pretty much how it worked.

Back then, networked CVS could be done only over a remote login program such as `rsh`. Using the same password for CVS access as for login access was an obvious security risk, and many organizations were put off by it. A major investment bank hired us to add a new authentication mechanism, so they could safely use networked CVS with their remote offices.

Jim and I took the contract and sat down to design the new authentication system. What we came up with was pretty simple (the United States had export controls on cryptographic code at the time, so the customer understood that we couldn't implement strong authentication), but as we were not experienced in designing such protocols, we still made a few gaffes that would have been obvious to an expert. These mistakes would easily have been caught had we taken the time to write up a proposal and run it by the other developers for review. But we never did so, because it didn't occur to us to think of the development list as a resource to be used to improve our contracted work. We knew that people were probably going to accept whatever we committed, and—because we didn't know what we didn't know—we didn't bother to do the work in a visible way, e.g., posting patches frequently, making small, easily digestible commits to a special branch, etc. The resulting authentication protocol was not very good, and of course, once it became established, it was difficult to improve, because of compatibility concerns.

The root of the problem was not lack of experience; we could easily have learned what we needed to know. The problem was our attitude toward the rest of the development community. We regarded acceptance of the changes as a hurdle to leap, rather than as a process by which the quality of the changes could be improved. Since we were confident that almost anything we did would be accepted (as it was), we made little effort to get others involved.

Obviously, when you're choosing a contractor, you want someone with the right technical skills and experience for the job. But it's also important to choose someone with a track record of constructive interaction with the other developers in the community. That way you're getting more than just a single person; you're getting an agent who will be able to draw on a network of expertise to make sure the work is done in a robust and maintainable way.

Update Your RFI, RFP and Contract Language

If you're hiring outside contractors to create software for you, the language you put in your Requests For Information (RFIs), Requests For Proposals (RFPs), and contracts becomes crucially important.

There is one key thing you must understand at the outset: the decision makers at most large-scale vendors don't really want their work to be open source. (The programming staff may feel differently, of course, but the path to the executive suite is usually smoother for those with an instinct for monopoly.) Instead, the vendors would prefer that a customer contract with them to produce bespoke software that, under the hood, shares many components with the *other* bespoke software they're producing for other customers.[4] That way the vendor can sell mostly the same product at full price many times. This is es-

[4]By the way, those common components are quite often open source libraries themselves. These days, it's typical for a proprietary software product to contain a lot of open source code, with a layer of proprietary custom code wrapped around the outside.

pecially true of vendors to government agencies, because the needs of government agencies are so similar, and because jurisdictional boundaries create an artificial multiplicity of customers who all have pretty much the same needs. Only minor customizations may be needed for each instance, but the different customers will pay full price each time.

As a customer, then, your starting point for a successful large-scale open source project is to set clear, explicit requirements about open source development from the beginning. From the RFI or RFP stage, all the way through the contract and into delivery and maintenance, you must require behaviors and deliverables that will result in a truly open source product — meaning, among other things, a product that has the potential to be supported and customized by vendors other than the one who originally developed it. Some of those requirements are:

- Design and development must be done in the open from the start of the project (see the section called “Be Open From Day One” in Chapter 2, *Getting Started*
- The code shall be licensed for open source distribution from the start of development through delivery and deployment.
- If the same vendor is both writing the software and deploying the production instances, require that deployed code must match the open source code. Don't let proprietary tweaks — and thus vendor lock-in — slip in via the back door.
- The product should have no dependencies on proprietary software modules; written permission from you must be obtained before any such dependencies are introduced.
- Documentation must be sufficient to allow third parties to understand, configure, and deploy the software.
- The vendor's engagement with third parties who become involved in the project should be described and budgeted for. If it is a successful open source project, there will eventually be community management overhead, so anticipate it: e.g., specify that the vendor must establish a participation workflow, review and prioritize contributions, etc.
- Describe the extent to which the vendor is expected to participate in publicity about the project, both among technical developer communities and among potential users.
- You, the customer, should be the copyright owner of the code written by the vendor.
- For any patents controlled by the vendor and affecting the project, there must be an unambiguous, non-restrictive patent grant not just to you but to everyone who receives the code.

Although this is not a complete list — every project is different — it should give you some idea of how to set expectations with your partners. The ability to recognize whether these expectations are being met, in spirit not just in letter, is also important of course, and is the subject of the next section.

IV&V: Use Third-Party Review Throughout Development

When a vendor who normally does proprietary development is hired to do open source development, by default the result is almost always be a product that is not truly open source and that no third party can actually deploy[5]. This section is about how to avoid that problem. While in some instances the vendor — or at least factions within the vendor — may be deliberately obstructive, more often the prob-

[5]While some selection bias no doubt informs my experience — after all, the consultant tends to get brought in when things are going wrong, not when they're going right — my assertion that proprietary vendors don't get open source right if left to their own habits is based not just on my own experiences but also on talking to many other people, who report the same finding with remarkable consistency.

lem is that they simply don't know what they don't know, and have no internal way to find out. Thus the most effective solutions is to bring in that knowledge from the outside: have a separate contract with a different company, one entirely independent of the primary vendor, to play the role of third-party open source participant.

There is a long tradition of such outside review in technical contracting. It's known as *IV&V*, for "*Independent Verification and Validation*"[6], and what keeps it independent is that the IV&V vendor is responsible to the customer, *not* to the primary development vendor. That part is crucial: even if the two vendors are contracting through the same prime vehicle, or one is a subcontractor to the other, it must be clear in the contracts that the IV&V function reports directly to the client, interacting with the primary development vendor only to *perform* the IV&V function.

The cost of IV&V review is much smaller than the cost of the main contract — generally, expect on the order of 5% to 10% — and the benefit is large: the difference between an end product that is not useably open source and one that is truly open source, able to be deployed and supported by anyone.

During development, the IV&V reviewers participate the way any third party would, posting in the project's public discussion forums, using the installation documentation to try to get the software up and running, reporting bugs via the public tracker, submitting pull requests, and so on. As the project reaches the alpha or beta stage, the reviewer confirms that the software can be deployed as documented, without reliance on proprietary dependencies or vendor-specific environmental conditions; that necessary per-deployment configurations can be made; that sample data can be loaded; that there exist documented paths by which third parties can participate in the project; and so on — in other words, that all the expectations one would have of an open source project are truly met.

But the reviewer's job is not just to review. The reviewer is there to *help* the primary vendor meet these expectations throughout development, and to report back to you, the customer, as to whether the vendor is doing so. In far too many cases, I have seen a nominally open source project be contracted for and developed, only for the customer to discover at the end — too late to do anything about it — that literally no party besides than the original vendor can actually deploy, maintain, or extend the software, because the vendor never came close to meeting normal open source standards. Had parallel, independent review been built into the process from the start, the problems would have been detected early, and an unsatisfactory outcome prevented. (Relatedly, see the section called “Be Open From Day One” in Chapter 2, *Getting Started*.)

Note that the primary vendor may often be quite unconscious of this failure. In their mind, they developed and delivered software the way they usually do, so what's the problem? The fact that no one other than them can deploy or modify the end result doesn't register as a failure, because in all their other projects that was never expected anyway. The fact that the contract requires it is meaningless unless the customer has some way to test and enforce that requirement. As most customers do not have the in-house technical capability to do so, without some kind of external review process the open source clauses in the contract are effectively void.

Independent review is not merely a sort of open source insurance, though it would be worthwhile even if it were only that. It is also an investment in the success of future partnerships with the primary vendor. The vendor becomes more inherently capable of performing quality open source work in the future, because the IV&V process provides a practical education in open source development. Thus, done right, third-party review results in both a healthier open source project and a healthier long-term relationship with the primary vendor. It also helps foster concentrations of expertise outside that primary contractor right from the start, as discussed in the section called “Foster Pools of Expertise in Multiple Places”. Ideally, at the end of development for a new open source product, you should have at least two independent commercial entities able to deploy and support the software: the primary development vendor, and

[6]For a more general discussion of IV&V, see en.wikipedia.org/wiki/Verification_and_validation [https://en.wikipedia.org/wiki/Verification_and_validation] and en.wikipedia.org/wiki/Software_verification_and_validation [https://en.wikipedia.org/wiki/Software_verification_and_validation]. Note that neither of those discusses open source specifically, however

the IV&V vendor. That's already twice as much supplier diversity as most projects have coming out of the gate, and it will be much easier to add a third vendor than a second.

Don't Surprise Your Lawyers

Corporate lawyers (and to a lesser degree lawyers in the non-profit world and in government) sometimes have an uneasy relationship with free software. They have often spent their careers diligently seeking to maximize the control and exclusivity their clients have over everything the clients produce — including software. A good lawyer will understand why their client is choosing to deliberately give up that control for some larger purpose, when it is explained, but even then may still be unfamiliar with the factors that go into choosing an open source license for the project, the interaction of the license with trademarks and patents, the legal technicalities of how to accept contributed code such that it can be redistributed, etc. (See Chapter 9, *Legal Matters: Licenses, Copyrights, Trademarks and Patents* for a deeper discussion of legal issues.)

The ideal course is to make sure your lawyers first understand *why* you are running an open source project, and give them a chance to familiarize themselves with open source in general, before you bring the particulars of the project to them. If the lawyers are good, they will know when they should seek help from outside counsel and will not hesitate to do so. By the time the project is under way, the lawyers should have enough familiarity with open source legal issues to make basic decisions with confidence, and to know when and where they need help.

Do not assume that open source is part of a standard legal education. It is not, at least as of this writing in 2015. If you wait until development is already under way and code is starting to be published before consulting your legal team, they may be forced to scramble and make under-researched decisions hastily. This will not be good for either the project or the organization, in the long run.

Funding Non-Programming Activities

Programming is only part of the work that goes on in an open source project. From the point of view of the project's participants, it's the most visible and glamorous part. This unfortunately means that other activities, such as documentation, formal testing, etc., can sometimes be neglected, at least compared to the amount of attention they often receive in proprietary software. Corporate organizations are sometimes able to make up for this, by devoting some of their internal software development infrastructure to open source projects.

The key to doing this successfully is to translate between the company's internal processes and those of the public development community. Such translation is not effortless: often the two are not a close match, and the differences can only be bridged via human intervention. For example, the company may use a different bug tracker than the public project. Even if they use the same tracking software, the data stored in it will be very different, because the bug-tracking needs of a company are very different from those of a free software community. A piece of information that starts in one tracker may need to be reflected in the other, with confidential portions removed or, in the other direction, added.

The sections that follow are about how to build and maintain such bridges. The end result should be that the open source project runs more smoothly, the community recognizes the company's investment of resources, and yet does not feel that the company is inappropriately steering things toward its own goals.

Quality Assurance (i.e., Professional Testing)

In proprietary software development, it is normal to have teams of people dedicated solely to quality assurance: bug hunting, performance and scalability testing, interface and documentation checking, etc. As a rule, these activities are not pursued as vigorously by the development community on a free software project. This is partly because it's hard to get highly-motivated labor for unglamorous work like test-

ing (committers have their names inscribed for all time in the history of the project, but there are fewer mechanisms for remembering the tester who found the bug a committer fixed), partly because people tend to assume that having a large user community gives the project good testing coverage, and, in the case of performance and scalability testing, partly because not all developers have access to the necessary hardware resources anyway.

The assumption that having many users is equivalent to having many testers is not entirely baseless. Certainly there's little point assigning testers for basic functionality in common environments: bugs there will quickly be found by users in the natural course of things. But because users are just trying to get work done, they do not consciously set out to explore uncharted edge cases in the program's functionality, and are likely to leave certain classes of bugs unfound. Furthermore, when they discover a bug with an easy workaround, they often silently implement the workaround without bothering to report the bug. Most insidiously, the usage patterns of your customers (the people who drive *your* interest in the software) may differ in statistically significant ways from the usage patterns of the Average User In The Street.

A professional testing team can uncover these sorts of bugs, and can do so as easily with free software as with proprietary software. The challenge is to convey the testing team's results back to the public in a useful form. In-house testing departments usually have their own way of reporting test results to their own developers, involving company-specific jargon, or specialized knowledge about particular customers and their data sets. Such reports would be inappropriate for the public bug tracker, both because of their form and because of confidentiality concerns. Even if your company's internal bug tracking software were the same as that used by the public project, management might need to make company-specific comments and metadata changes to the tickets (for example, to raise an ticket's internal priority, or schedule its resolution for a particular customer). Usually such notes are confidential—sometimes they're not even shown to the customer. But even when they're not confidential, they're not very helpful to the public project.

Yet the core bug report itself *is* important to the public. In fact, a bug report from your testing department is in some ways more valuable than one received from users at large, since the testing department probes for things that other users won't. Given that you're unlikely to get that particular bug report from any other source, you definitely want to preserve it and make it available to the public project.

To do this, either the QA department can file tickets directly in the public ticket tracker, if they're comfortable with that, or an intermediary (usually one of the developers) can "translate" the testing department's internal reports into new tickets in the public tracker. Translation simply means describing the bug in a way that makes no reference to customer-specific information (the reproduction recipe may use customer data, assuming the customer approves it, of course).

It is definitely preferable to have the QA department filing tickets in the public tracker directly. That gives the public a more direct appreciation of your company's involvement with the project: useful bug reports add to your organization's credibility just as any technical contribution would. It also gives developers a direct line of communication to the testing team. For example, if the internal QA team is monitoring the public ticket tracker, a developer can commit a fix for a scalability bug (which the developer may not have the resources to test herself), and then add a note to the ticket asking the QA team to see if the fix had the desired effect. Expect a bit of resistance from some of the developers; programmers have a tendency to regard QA as, at best, a necessary evil. The QA team can easily overcome this by finding significant bugs and filing comprehensible reports; on the other hand, if their reports are not at least as good as those coming from the regular user community, then there's no point having them interact directly with the development team.

Either way, once a public ticket exists, the original internal ticket should simply reference the public ticket for technical content. Management and paid developers may continue to annotate the internal ticket with company-specific comments as necessary, but use the public ticket for information that should be available to everyone.

You should go into this process expecting extra overhead. Maintaining two tickets for one bug is, naturally, more work than maintaining one ticket. The benefit is that many more coders will see the report and be able to contribute to a solution.

Legal Advice and Protection

Corporations, for-profit or nonprofit, are almost the only entities that ever pay attention to complex legal issues in free software. Individual developers often understand the nuances of various open source licenses, but they generally do not have the time or resources to competently handle legal issues themselves. If your company has a legal department, it can help a project by assisting with trademark issues, copyright license ownership and compatibility questions, defense against patent trolls, etc. If the project decides to organize formally, or to join an existing umbrella organization, your legal department can help with issues of corporate law, asset transfer, reviewing agreements, and other due diligence matters.

Some more concrete ideas of what sorts of legal help might be useful are discussed in Chapter 9, *Legal Matters: Licenses, Copyrights, Trademarks and Patents*. The main thing is to make sure that communications between the legal department and the development community, if they happen at all, happen with a mutual appreciation of the very different universes the parties are coming from. On occasion, these two groups talk past each other, each side assuming domain-specific knowledge that the other does not have. A good strategy is to have a liaison (usually a developer, or else a lawyer with technical expertise) stand in the middle and translate for as long as needed.

Documentation and Usability

Documentation and usability are both famous weak spots in open source projects, although I think, at least in the case of documentation, that the difference between free and proprietary software is frequently exaggerated. Nevertheless, it is empirically true that much open source software lacks first-class documentation and usability research.

If your organization wants to help fill these gaps for a project, probably the best thing it can do is hire people who are *not* regular developers on the project, but who will be able to interact productively with the developers. Not hiring regular developers is good for two reasons: one, that way you don't take development time away from the project; two, those closest to the software are usually the wrong people to write documentation or investigate usability anyway, because they have trouble seeing the software from an outsider's point of view.

However, it will still be necessary for whoever works on these problems to communicate with the developers. Find people who are technical enough to talk to the coding team, but not so expert in the software that they can't empathize with regular users anymore.

A medium-level user is probably the right person to write good documentation. In fact, after the first edition of this book was published, I received the following email from an open source developer named Dirk Reiners:

> One comment on Money::Documentation and Usability: when we had some money to spend and decided that a beginner's tutorial was the most critical piece that we needed we hired a medium-level user to write it. He had gone through the induction to the system recently enough to remember the problems, but he had gotten past them so he knew how to describe them. That allowed him to write something that needed only minor fixes by the core developers for the things that he hadn't gotten right, but still covering the 'obvious' stuff devs would have missed.
>
> His case was even better, as it had been his job to introduce a bunch of other people (students) to the system, so he combined the experience of many people, which is something that was just a lucky occurrence and is probably hard to get in most cases.

Funding User Experience (UX) Work

The field of user experience (*UX*) design has lately (as of this writing in early 2014) begun to acquire a new seriousness of purpose and consistency of professional standards. Naturally, one thing many companies think of when they want to help improve an open source project is to fund UX work, since that's just the sort of thing that projects often overlook or, in some cases, don't even know they need.

As with many other types of engagement, do not assume that a UX expert can be parachuted into the project. User experience design is not a checkbox. It is an attitude taken by a team throughout development, and one of the primary qualifications to look for in UX contractors is their ability to gain long-term credibility with the developers, and to help developers pay attention to user experience goals. For example, in addition to their innate domain knowledge, UX designers often know how to set up and incorporate feedback from user trials — but those trials will only be effective if the results are presented to the development team in a way that makes it easy for the developers to take the results seriously. This can only happen through a sustained, two-way interaction, in which UX experts are subscribed to the appropriate project forums and take the attitude that they are a kind of specialized developer on the project, rather than an outside expert providing advice. Use UX experts who have worked with open source projects before, if possible.

Providing Hosting/Bandwidth

The inexorable rise of zero-cost canned hosting sites (see the section called “Canned Hosting” in Chapter 3, *Technical Infrastructure*) for open source projects has meant that it is increasingly unnecessary for projects to get corporate support for basic project-hosting infrastructure. It still happens sometimes, usually in cases where the company itself started the project, and is trying to create or keep an association in the public's mind between the project and the company. The most common technique for creating this association is for the company to host the project's resources under the company's domain name, thus getting association through the project's URLs.

While this will cause most users to think of the software as having *something* to do with your company, it can also cause a problem: developers are aware of this associative tendency too, and may not be very comfortable with having the project hosted under your domain unless you're seriously contributing to the project—not just bandwidth and server space, but significant amounts of development time. After all, there are a lot of free places to host these days. The community may eventually feel that the implied misallocation of credit is not worth the minor convenience brought by donated hosting, and may even attempt to take the project elsewhere. So if you want to provide hosting, do so—but if you are actually trying to create a public association between your company and the project, make sure that the level of support you provide is matched to the amount of credit you claim, whether you claim it via URL, banner ads, or some other means.

Providing Build Farms and Development Servers

Many projects have infrastructure needs beyond just hosting of code, bug tracker, etc. For example, projects often use *continuous integration* (CI) testing (a.k.a. *build farms*) to automatically ensure that the changes developers are committing both integrate into the mainline trunk and pass all regression tests[7]. However, depending on the size and complexity of the codebase, the number of developers checking in changes, and other factors, running a responsive build farm can cost more money than any individual developer has at their disposal. A good way to help, and gain some goodwill in the process, is to donate the server space and bandwidth *and* the technical expertise to set up the continuous integration and automated testing. If you don't have the technical expertise available on staff, you could hire someone from the

[7]The Wikipedia page en.wikipedia.org/wiki/Continuous_integration [https://en.wikipedia.org/wiki/Continuous_integration] has a good description of this practice and its variants.

project to do it, or at the very least give some of the project's developers administrative access to the CI servers so they can set things up themselves.

Sponsoring Conferences, Hackathons, and other Developer Meetings

A very effective use of funds is to sponsor in-person contact between developers who might not otherwise meet. The usefulness of in-person meetings — e.g., conferences, hackathons, smaller informal meetups, etc — is mainly discussed in the section called “Meeting In Person (Conferences, Hackfests, Code-a-Thons, Code Sprints, Retreats)” in Chapter 8, *Managing Participants*. Here I will simply mention that encouraging such encounters is a very good use of money in an open source project. From a corporate sponsorship point of view, nothing creates good will like a plane ticket and a hotel room. From a personnel management point of view, it is healthy for your own developers to have in-person contact with their peers in the open source projects they work on, and when those peers work at at other companies, project-centric meetups are the perfect neutral ground for such meetings.

Sending your developers to conferences is also a good way to signal commitment to a project. When others meet your developers at a conference the first time, it is a signal that your company has a real investment in the project. But when your developers show up again at the same conference the next year, still working on the same project, that's a very powerful signal that your organizational commitment to the project is long-term and strategic. This gives your developers an advantage in influencing the direction of the project, because they are seen as people who will be around for the long term, and it of course gives your company a recruiting advantage when you are looking for new developers to work on the same project.

Even when you don't have people traveling to a meetup, you can still sponsor some of the meetup's expenses. Everyone remembers fondly the company that sponsors the pizza, or lunch, or drinks or dinner for one night of the meetup.

Marketing

Although most open source developers would probably hate to admit it, marketing works. Good marketing *can* create buzz around an open source product, even to the point where hardheaded coders find themselves having vaguely positive thoughts about the software for reasons they can't quite put their finger on. It is not my purpose here to dissect the arms-race dynamics of marketing in general. Any corporation involved in free software will eventually find itself considering how to market themselves, the software, or their relationship to the software.

Much of the advice in this section is simply about how to avoid common pitfalls in marketing open source products (see also the section called “Publicity” in Chapter 6, *Communications*), although we will start by examining a major marketing advantage that open source products enjoy over proprietary products, and that open source businesses should promote as often as possible: the lack of vendor lock-in.

Open Source and Freedom from Vendor Lock-In

Vendor lock-in is what happens when a vendor sells a service or product to a customer, perhaps at a cheap up-front price, but the customer has to make certain further investments in order to *use* the product — e.g., infrastructure changes, workflow and other process changes, data reformatting, retraining, etc. The cost to the customer of switching *away* from that vendor's product is now the strength with which the vendor has the customer locked in. Even if the customer is eventually unhappy with the vendor, by that point the total cost of moving to someone else may be quite high, and that cost is independent of whatever licensing or service fees the vendor charges.

The great commercial strength of open source is that product and vendor are not the same. In open source, you can switch to another vendor, or to a combination of vendors, or even a combination of vendor and in-house support, all while continuing to use the same product in more or less the same way.

So if you sell open source, make sure your potential customers are clear on this point, and give them as many concrete examples as you can. It may, in some circumstances, even be useful to point out the existence of some of your competitors, because their presence paradoxically reassures the customer that choosing you is a safe decision — if things don't work out, there are other options. If you just make sure things work out, then the customer will never need to seek out those other options.

Proprietary vendors often compete against open source by talking about the "*total cost of ownership*", that is, they sell against open source's up-front cost of zero — no per-copy royalties, no per-seat license fees — by pointing out, reasonably enough, that although there may be no licensing fees, in practice software integration involves organizational and technical costs that can be quite significant. This is quite true, as far as it goes. What they don't say is that this argument works against them as much as it works for them: to the extent that there *are* such costs — and there really are — the danger to the customer of vendor lock-in is directly proportional to them. I would love to see open source sales representatives talk more about the "*cost of total ownership*", that is, how much does it cost a company to be totally owned by its software vendors? With open source, customers are not owned — they are the owners, to exactly the degree that they want to be, and they can outsource as much of that responsibility to outside vendors as they want. Their relationships with those vendors are thus more likely to be based on mutual satisfaction and mutual benefit, not on an asymmetrical monopoly that gives existing vendors undue inertia in customers' procurement decisions.

Remember That You Are Being Watched

For the sake of keeping the developer community on your side, it is *very* important not to say anything that isn't demonstrably true. Audit all claims carefully before making them, and give the public the means to check your claims on their own. Independent fact checking is a major part of open source, and it applies to more than just the code.

Naturally no one would advise companies to make unverifiable claims anyway. But with open source activities, there is an unusually high quantity of people with the expertise to verify claims—people who are also likely to have high-bandwidth Internet access and the right social contacts to publicize their findings in a damaging way, should they choose to. When Global Megacorp Chemical Industries pollutes a stream, that's verifiable, but only by trained scientists, who can then be refuted by Global Megacorp's scientists, leaving the public scratching their heads and wondering what to think. On the other hand, your behavior in the open source world is not only visible and recorded, it is also easy for many people to check it independently, come to their own conclusions, and spread those conclusions by word of mouth. These communications networks are already in place; they are the essence of how open source operates, and they can be used to transmit any sort of information. Refutation is usually difficult, if not impossible—especially when what people are saying is true.

For example, it's okay to refer to your organization as having "founded project X" if you really did. But don't refer to yourself as the "makers of X" if most of the code was written by outsiders. Conversely, don't claim to have a deeply involved, broad-based developer community if anyone can look at your repository and see that there are few or no code changes coming from outside your organization.

Case study: You can't fake it, so don't try

Once I saw an announcement by a very well-known computer company, stating that they were releasing an important software package under an open source license. When the initial announcement came out, I took a look at their now-public version control repository and saw that it contained only three revisions. In other words, they had done an initial import of the source code, but hardly anything had hap-

pened since then. That in itself was not worrying—they'd just made the announcement, after all. There was no reason to expect a lot of development activity right away.

Some time later, they made another announcement. Here is what it said, with the name and release number replaced by pseudonyms:

> *We are pleased to announce that following rigorous testing by the Singer Community, Singer 5 for Linux and Windows are now ready for production use.*

Curious to know what the community had uncovered in "rigorous testing," I went back to the repository to look at its recent change history. The project was still on revision 3. Apparently, they hadn't found a *single* bug worth fixing before the release! Thinking that the results of the community testing must have been recorded elsewhere, I next examined the bug tracker. There were exactly six open tickets, four of which had been open for several months already.

This beggars belief, of course. When testers pound on a large and complex piece of software for any length of time, they will find bugs. Even if the fixes for those bugs don't make it into the upcoming release, one would still expect some version control activity as a result of the testing process, or at least some new tickets. Yet to all appearances, nothing had happened between the announcement of the open source license and the first open source release.

The point is not that the company was lying about the "rigorous testing" by the community (though I suspect they were). The point is that they were oblivious to how much it *looked* like they were lying. Since neither the version control repository nor the ticket tracker gave any indication that the alleged rigorous testing had occurred, the company should either not have made the claim in the first place, or should have provided a clear link to some tangible result of that testing ("We found 278 bugs; click here for details"). The latter would have allowed anyone to get a handle on the level of community activity very quickly. As it was, it only took me a few minutes to determine that whatever this community testing was, it had not left traces in any of the usual places. That's not a lot of effort, and I'm sure I'm not the only one who took the trouble.

Transparency and verifiability are also an important part of accurate crediting, of course. See the section called “Credit” in Chapter 8, *Managing Participants* for more on this.

Don't Bash Competing Open Source Products

Refrain from giving negative opinions about competing open source software. It's perfectly okay to give negative *facts*—that is, easily confirmable assertions of the sort often seen in good comparison charts. But negative characterizations of a less rigorous nature are best avoided, for two reasons. First, they are liable to start flame wars that detract from productive discussion. Second, and more importantly, some of the developers in *your* project may turn out to work on the competing project as well. This is more likely than it at first might seem: the projects are already in the same domain (that's why they're in competition), and developers with expertise in that domain may make contributions wherever their expertise is applicable. Even when there is no direct developer overlap, it is likely that developers on your project are at least acquainted with developers on related projects. Their ability to maintain constructive personal ties could be hampered by overly negative marketing messages.

Bashing competing closed-source products seems to be more widely accepted in the open source world, especially when those products are made by Microsoft. Personally, I deplore this tendency (though again, there's nothing wrong with straightforward factual comparisons), not merely because it's rude, but also because it's dangerous for a project to start believing its own hype and thereby ignore the ways in which the competition may actually be superior. In general, watch out for the effect that marketing statements can have on your own development community. People may be so excited at being backed by marketing dollars that they lose objectivity about their software's true strengths and weaknesses. It is normal, and even expected, for a company's developers to exhibit a certain detachment toward market-

ing statements, even in public forums. Clearly, they should not come out and contradict the marketing message directly (unless it's actually wrong, though one hopes that sort of thing would have been caught earlier). But they may poke fun at it from time to time, as a way of bringing the rest of the development community back down to earth.

Don't Bash Competing Vendors' Developers

Another situation companies find themselves in, when selling services based on open source software, is that they have competitors in the marketplace who may be selling services based on the *same* software.

If you're going to sell your company's services, you inevitably will need to compare your company against others selling the same or similar things. This is expected, and in many ways healthy. However, be careful to avoid straying into public criticism of the other development teams or even, to a degree, of their development priorities. Your own developers have to work directly with those competitors' developers in the open source project, and they often have friendly relations, show up at the same conferences, etc. Even if that's not the case today, it may be tomorrow. Furthermore, you may find yourself *hiring* developers from your competitors; if you burn up available goodwill in advance, you may not get the best candidates.

Without mentioning names, in part because the situation eventually got better and I don't want to rekindle the flames now, I will say that I saw exactly this happen between two companies (one of whom was my employer at the time) who were competing to sell services based on the same open source software. The ill will stirred up among the project's developers by the marketing statements of one company (not my employer) had real consequences, and that company lost out on retaining the services of some excellent developers because it failed to think through the fact that their marketing in the commercial realm was also visible and had effects in the development community.

"Commercial" vs "Proprietary"

One common pattern among companies involved in open source software is to market a fully open source version of their product alongside, and in direct comparison to, an enhanced proprietary version. Since the open source version is free software, you *could* in theory add those enhancements yourself, or collaborate with others to do so, but in practice, the effort required to do that (and to maintain a now separate fork of the project) is, for each collaborator, much greater than the cost of just paying for the proprietary version, so it rarely happens.

This sales model is often referred to as "open core", that is, a core set of functionality that is available as open source software, with a more featureful application wrapped around it as proprietary software. This usually depends on the core open source code having a non-copyleft license, of course, and is discussed in more detail in the section called “Proprietary Relicensing” in Chapter 9, *Legal Matters: Licenses, Copyrights, Trademarks and Patents*.

Open core is somewhat controversial among open source developers, but it has been successful strictly from a business point of view: companies that do it make money in the way that they expect to make money. However, there is bit of marketing slippage that many of these companies fall into, and I would like to point it out here so I can try to convince you not to be part of the problem.

If you sell a free software version and an enhanced proprietary version of your product, please use the words "open source" and "proprietary" to refer to them, respectively. Do *not* call the open source version the "Community Edition" and the proprietary version the "Commercial Edition" (or "Enterprise Edition"). Aside from the fact that everyone knows there is very little "community" around these so-called "Community Editions", there is a deeper problem here. Calling the proprietary version the "Commercial Edition" implies that open source software is not commercial, which is completely untrue; calling it the "Enterprise Edition" implies that open source software is not suitable for enterprise-level use. The former is untrue because open source software is commercial by definition: the license guarantees the

freedom to use the software for any commercial purpose. (Open source is *anti-monopoly*, of course, but that's separate from commerciality.) The latter is demonstrably untrue: open source is widely used at enterprise scale, with and without third-party support.

This kind of misleading marketing particularly hurts efforts by open source companies to get their software accepted by governments and by other buyers who have sophisticated procurement requirements. These procurement regulations often include stipulations that purchased software must be "commercial", "commercial off-the-shelf", or "commercially available" — definitions that all open source software meets — and portraying open source as non-commercial gives purchasing officers a misimpression. The extent to which those decision-makers think of open source as inherently non-commercial hurts open source software as a whole, by stymieing those who are doing their best to make inroads in these kinds of procurement environments.

Open Source and the Organization

Through the consulting work I've done in the years since the first edition of this book was published, it's become clear to me that there are special concerns that apply to organizations launching or participating in open source projects. Organizations contain formal management structures and informal social structures: both are affected by engagement with open source projects, and both sometimes need to be changed to better support open source activity by the individuals within the organization. In particular, government agencies have special pitfalls to watch out for when working with open source projects.

This section therefore examines organizational issues generally, and some issues specific to government agencies, and offers some advice about how to make organizational engagement with open source more likely to succeed. Many of these recommendations will be brief and somewhat generalized, not because there isn't more depth to go into, but because the specifics can vary so much from organization to organization that exploring all the possibilities here would require too much space. Please treat these bits of advice as starting points, not as complete recipes in themselves.

Dispel Myths Within Your Organization

In organizations that have been producing or using proprietary software for a long time, certain myths about open source software sometimes circulate. One traditional source of such myths is, of course, sales representatives from vendors of proprietary systems. But one can't attribute it all to them. It's just as often the case that someone had some bad experiences in an open source project, or used open source in the past without ensuring proper support channels, and since that was their first experience in an unfamiliar territory, the entire territory is now tainted.

Here are some of the myths I've encountered most frequently:

If it's open, that means anyone can change our code.

> Believe it or not, you need to be prepared to respond to this. Sometimes people — particularly senior decision-makers who have limited technical experience — don't understand the difference between an upstream codebase allowing anyone to copy the code and modify their own copies, and someone modifying the *particular instance that you deploy*. The former is just the definition of open source, of course; the latter would be a security vulnerability if it happened, but it has nothing to do with the license on the code. I mention this myth merely to prepare you for encountering it, because otherwise you might not expect that anyone could hold this particular misunderstanding. Trust me, they *can*, and you need to be ready to answer it.

Open source software is insecure, because anyone can change the code.

> This is so easy to answer that I won't give a detailed refutation here; again, I merely note it so you can be prepared for it. If you find yourself having to explain why open source software is at least

as secure as any other kind of software, if not more secure, you may wish to use the excellent resources provided by Dr. David A. Wheeler at dwheeler.com/#oss [http://www.dwheeler.com/#oss], and the arguments at mil-oss.org/learn-more/security-model-misconceptions [http://www.mil-oss.org/learn-more/security-model-misconceptions].

Open source comes with no support.

There are plenty of companies that sell support for open source software, and they're not hard to find. There are also wonderfully helpful unofficial support communities on the Internet for different open source packages, of course, but often what organizations are looking for is a phone number to call, or a support chat room, that has a guaranteed response time. These are available, it's just that the source from which you procure the software may be unrelated to the source from which you procure the support. The best way to respond to this myth is to ask specifically for what packages support is desired, and then show some sources of support available for them.

Open source is cheaper.

Licensing costs are often not the largest cost with proprietary software; they are often outweight by training costs, installation and configuration costs, and other factors that make up the "total cost of ownership". But all of those other costs are, on average, the same for open source software. Don't make the mistake of pitching your organization on open source software on the grounds that it is cheaper. At least in terms of the most easily quantified costs, it is not. It is often cheaper in the long run, because it frees your organization from proprietary vendor lock-in (see the section called “Open Source and Freedom from Vendor Lock-In”), reduces training costs for new employees (because they arrive already familiar with the software), gives you greater ability to customize software to your needs, etc. But these are long-term benefits, and they may not show up directly on a balance sheet unless you take steps to make your accounting reveal them. In the short term, open source generally isn't cheaper than proprietary software, and shouldn't be pitched that way.

If we open source this project, we'll have to spend a lot of time interacting with outside developers.

You open source your code, not your time and attention. You are never under any obligation to respond at all to outside parties, let alone engage substantively with them. You should only do so when engaging will benefit *you* — which it often will; after all, one of the key strengths of open source is that it enlarges the collective brain of your development team in direct proportion to how much they interact with other developers who become interested in the code. But that engagement is always under your control and at your discretion. If you don't want your team's attention going to bug reports or development questions from the outside, that's fine. Just be up front about that in project announcements and in the documentation, so that others can take that into account before they put a lot of energy into trying to communicate with your developers, and so they can decide whether forking to create a more open community would make sense for them (indeed, sometimes it might even be to your advantage for them to do that).

If we open source this project, then we'll have to release all our other stuff as open source too.

This myth usually results from a misunderstanding of copyleft licenses and the GNU General Public License (GPL) in particular. I won't go into detail here; see Chapter 9, *Legal Matters: Licenses, Copyrights, Trademarks and Patents* for a discussion of what the GPL actually stipulates. After reading that chapter, especially the section called “The Copyright Holder Is Special, Even In Copyleft Licenses”, you will be able to explain why this belief is incorrect.

Developers will devote attention to this code just because we released it.

People with little experience in open source sometimes assume that the mere act of releasing code to the public will result in a flurry of attention from other developers — questions, patches, high-

quality code review, bug reports, etc. But what actually happens, in most cases, is silence. Most good developers are busy people, and they're not going to pay attention to your project until they have some reason to. If your code is good and solves a real problem, you can expect word to travel to the right places eventually, and of course you can help that word along with tactically smart announcements and posts (see the section called “Publicity” in Chapter 6, *Communications*). But until your code has had time to naturally accumulate credibility and mindshare, most people won't pay any attention, so you shouldn't expect that first release to be a big deal for anyone but you.

There is a situation in which this myth is not a myth. A large organization with a reputation and a dedicated public relations team *can* create buzz around an initial open source release. If you do this, then make sure not to squander that buzz: be ready to constructively engage the developer attention you attract right away.

Other companies / cities / whatever will pick up this software and start using it right away.

Adopting any software involves costs. Indeed, merely *evaluating* software involves costs. So when you release a new open source project that you and your team are excited about, that doesn't necessarily mean other entities are going to pick it up right away and run with it. Many of them may notice it, if you've done your announcement process well, but but that just means they'll put it on their list of things to investigate based on organization-wide priorities — in other words, they'll take a closer look based on *their* schedule, not yours. So don't expect a flood of early adopters. You may get a few, and they should definitely be cultivated because they will provide the word-of-mouth that gets you more adopters. But in general you're more likely to see a trickle of early adopters over the first year or so after your initial release, than to see a flood of them immediately when the release is made.

Foster Pools of Expertise in Multiple Places

Sometimes organizations that are accustomed to procuring proprietary software treat open source software as if it were proprietary, in the sense that they assume there is exactly one authoritative provider of expert support, and that therefore it is necessary to have a commercial relationship with that provider.

That's not how open source works. One of the great strengths of open source is the availability of support from multiple, competing providers. It's perfectly fine, and often advisable, to have a commercial relationship with just one of those sources, but you must remember that support in open source is fundamentally a *marketplace*, not an add-on feature that just happens to come with the software license, as is often the case with proprietary software. Actually, even proprietary software sometimes has a competitive support marketplace — think for example of the third-party support providers for Oracle databases and Microsoft operating systems — but in open source these marketplaces tend to be more fluid and not as dominated by single, easily-recognizeable giants, because there isn't necessarily one commercial outfit that automatically assumes a place at the top of the hierarchy to sell gold-label support (e.g., as Oracle or Microsoft themselves would be, in the example just given).

Ideally, the goal of fostering independent pools of expertise should even affect how you structure contracts to develop the software in the first place. If you hire a firm to develop new open source software, have a few of your own programmers working alongside them if possible, so that you accumulate some in-house expertise. This is not necessarily because you won't want to use the same firm for future maintenance — they might be a great choice — but just so that you'll have a better bargaining position and not be locked in. Essentially, the more people *in different organizations* who know the code, the better position that code is in, and the better position you are in. This is also one of the side benefits of holding hackathons, as discussed in the section called “Sponsoring Conferences, Hackathons, and other Developer Meetings”.

If your organization does not have enough in-house technical ability to participate in the development process directly alongside your contractor, or at least to perform knowledgeable review, then I strongly

recommend finding a third-party to provide *independent* deployability and maintainability review while the project is under way, as described in the section called “IV&V: Use Third-Party Review Throughout Development”.

Establish contact early with relevant communities

Another way to foster independent sources of expertise is to establish contact with potentially interested technical communities early and often during development. They're almost always out there. For example, if you're developing software with geospatial functionality, there is an open source geospatial community that probably wants to hear about it; if you're developing software to process financial data, or medical data, there are open source *fintech* and medical data communities.

You may even have already announced your project to those people when you began, as discussed in the section called “Announcing” in Chapter 2, *Getting Started*. But there's more you can do to create external reservoirs of knowledge. When your project runs across a design issue that you suspect others may have encountered before, it's fine to ask them how they handled it, as long as you do your homework by first finding out what you can from their code and documentation and then asking any remaining questions. You can also arrange small-scale contracts with developers who are active in related projects, to serve two goals at once: improving your project's quality while establishing mindshare in places that may be strategically useful later.

Don't Let Publicity Events Drive Project Schedule

Although open source projects are amenable to software project management techniques, in general if you have an active developer community you do lose some control over the exact timing of events in the life of the project, especially the scheduling of releases. Or rather, you can still have as much control as you want, but now there are other things you can lose if you exercise that control in the wrong way. For example, if the release manager (see the section called “Release manager” in Chapter 7, *Packaging, Releasing, and Daily Development*) is someone from outside your organization, and she's doing a good job, then if you try to force the release to be on a certain precise date, you may cause her and many of the developers participating in release-specific work to give up and devote their attention to something else. You'd gain fine-grained control of the release schedule, but at the cost of lower quality releases and the possible loss of some of your development community.

This is just one example illustrating the general principle that if you have publicity needs related to an open source project, you shouldn't let those needs drive the project's schedule. If you arrange a press conference for the project reaching 1.0 and being deployed live, but then the developers decide on an extra two weeks of testing because of some last-minute bugs, you'll have some improvising to do. (This example is drawn from real life, by the way.)

There are two ways to achieve this independence, and they are not mutually exclusive. One way is to just let project events drive publicity instead of the other way around, such as by preparing release announcements ahead of time but being ready to publish them based on when the release is actually done. The other way is to create publicity events that are not bound to development milestones, but are rather associated with project-related things that *are* able to be scheduled, such as conference appearances, hackathons, major deployments, etc.

You might be tempted to try a third way: to bring the development community into the scheduling process, so that through consensus you are able schedule certain milestones accurately enough to tie timed publicity to them. While that may sound like a good idea, in practice it rarely works. An open source development community's first priority is the software itself, and making sure it meets the needs its participants are working toward. Of course the community wants releases and other deadlines to be met with reasonable regularity, and every development community makes tradeoffs for that. But even

with the best of intentions among all parties, you can never guarantee how that tradeoff will be decided in a particular case, when things get down to the wire. The outcome of a community's decision-making process cannot be anticipated with perfect accuracy, by definition — if it could, there would be no need for a decision-making process. So while it's fine to try to influence the community's priorities in ways that work to your advantage, you should avoid relying on that for scheduling purposes, because you won't succeed at it every time.

The Key Role of Middle Management

If you intend to have long-term organizational engagement with open source software projects, the people in your middle layer of management will play a key role in determining whether you succeed or fail. Supervising programmers who spend part or all of their time on open source projects is more complex than supervising programmers on purely internal projects. Many aspects of the developers' work and schedule will be strongly influenced by external factors not under the control of management, and in any case the developers' own desires may not always perfectly line up with the employer's. After all, each developer now has two unrelated audiences to satisfy: her employer, as embodied by her direct manager, and her colleagues in the open source project, many of whom may work for other employers. If a manager is not sufficiently sensitive to this dynamic, then developers can start to feel as though they serve two conflicting masters. Sometimes this conflict is the result of poor planning, and other times it is real and unavoidable. Good management can prevent the former situation from happening in the first place, and in the latter situation, good management is essential for recognizing it and addressing it in some way that gives the developer clarity and a way to handle the conflicted situation.

Middle managers also have not only the usual upward and lateral internal reporting responsibilities, but are to some degree responsible for the image — the open source brand identity — of the organization in the projects where its developers participate. This is a lot like having an entire extra constituency to satisfy, and managers who have no experience with open source participation themselves are unlikely to have a solid understanding of how to position the organization and its developers within the project.

Furthemore, the middle layer of management is often best positioned to serve as a communications conduit and information filter between the project (that is, the whole project including all its other participants) and the company. The wealth of information available from the activity in an open source project is most useful to the organization if there is a filtered channel by which the most interesting activities can be communicated to the relevant stakeholders within the organization — stakeholders who might include other technical staff, executives, and sales team members. But both by their position and their temperament, the programmers themselves are often not best suited to serve as this conduit. They may have a very deep understanding of the particular projects they work on, but they often have a less complete view of the organization's interests — for example, in a commercial environment, the programmers often do not have a clear idea of how the project fits into the company's various lines of business or into its sales processes. Middle managers are better positioned to maintain the requisite bidirectional sensitivity: aware enough of the project to ask the programmers for more information when necessary, and aware enough of the organization to have a sense of what in the project might be unexpectedly relevant to the organization.

Think carefully about who occupies the middle management positions that serve as the interface between the organizations's priorities and the open source project's development direction, and provide them with extra training if necessary. It is best if the managers themselves have had direct experience as participants in some open source project. This doesn't have to be the same project as the one for which they are now managing developers; the situations and tensions that arise in open source projects tend to be similar, so experience from one project will generally translate well to other projects. But a manager who has never dealt with open source projects first-hand at all will start with limited ability to understand the various pressures faced by the organization's developers operating in open source environments, and limited ability to play the role of communications conduit between the organization and the project.

Innersourcing

Innersource or *innersourcing* means using standard open source development practices only within the boundaries of an organization. For example, a company might move all of its projects to GitHub (albeit in private, not public, repositories), and declare that, inside the company, any engineer can report bugs and contribute pull requests to any project anywhere in the company. Innersourcing also often includes sincere efforts at internal cultural change: managers encouraging developers to speak their mind on both technical and process issues, developers being given more latitude to choose which projects and teams they work with, etc.

In early 2016 I conducted interviews[8] with open source specialists at a number of medium- and large-sized technology companies, many of whom had observed innersourcing efforts and were willing to talk about the results. What they reported was pretty consistent from company to company, and consistent with my own experience as a consultant: innersourcing really *can* make a positive difference, in several ways, but it's also definitely not the same as true open source.

For companies that already participate in open source projects, innersourcing can reduce the difference between internal development practices and external ones. If some of your engineers have to participate in upstream open source projects anyway, in which they must use typical open source collaboration tools and adhere to open source standards for submitting and reviewing code and documentation, then moving the company's internal engineering infrastructure and conventions in that direction means less context-switching overhead for existing staff, an easier onboarding process for new hires, and sometimes improved technical compatibility between internal and external projects. (For these reasons, innersourcing is also often used as the first "baby steps" toward genuine corporate participation in open source projects).

But the benefits of innersourcing go beyond that. When accompanied by a real commitment to reduce managerial and organizational barries to engineers following their own lead in contributing to projects across the company, innersourcing can improve both morale, help spread expertise around the company more effectively (because expertise will come closer to flowing along its natural paths, rather than being directed by management), and make software development more efficient.

Nevertheless, innersource is not the same as open source, nor is it even "open source lite". The managers we talked to reported that innersourced projects have less of the uncontrolled dynamics of truly open source projects, because all the actors in innersourcing are, ultimately, all part of the same hierarchical authority structure. Fundamentally, open source dynamics require at least the potential for totally permissionless modification (i.e., you don't have to worry what someone might think of a fork). But when software only circulates within a given management hierarchy, then that potential for permissionless collaboration vanishes — and with it, the potential for true open source behavior vanishes too. The permission structure that governs one's behavior with respect to the code is not just a matter of the code's license: it's also about whom you report to, what others in the hierarchy might think about your changes, etc.

In the long run, the dynamics of open source collaboration require an external supply of freedom. There must always be people who could, in principle, fork or do whatever they want without worrying about consequences to the original authors' organization. When that external freedom is removed, everything changes.

Innersourcing also fails the "portable résumé" test — an employee can't take the code with her, and her work will not be publicly visible (see the section called “Hiring Open Source Developers”.). She can be alienated from her work, if she leaves the company, which means that her motivation to personally invest is reduced.

[8] Actually, my friend and business partner James Vasile and I both conducted these interviews, and we were much aided by O'Reilly Media providing introductions to open source staff at a few companies where we did not have personal contacts.

None of this means that innersourcing isn't worth it. It can be very beneficial on its own terms, and is also sometimes useful as an intermediate step for a traditionally closed company that's still figuring out how to do open source participation. Just don't imagine that innersourcing is somehow "just like open source, but inside our company". They're two different things, and shouldn't be confused.

Hiring Open Source Developers

If you're trying to hire developers who have open source experience, you have a big advantage compared to hiring other kinds of developers. Most of the résumé of an open source developer is public — it's everything they've ever done in every open source project they've ever worked on, because all of that activity is publicly archived.[9] But you shouldn't need to go searching for all of it. When you put out a job posting, tell prospective candidates directly that the résumé they send in should include references to their open source profile. This means their committer accounts on the projects where they've been active (or their account names at the overall project hosting sites where they're been active, e.g., their GitHub.com username), the email addresses or usernames they have used when posting in discussion forums, documentation they have written, and anything else that would lead you to places where you can see their open source project activity.

The kind of activity to look for is not only their direct technical activity, but also their relations with the other developers in the project. So look at the candidates commits, but look also at the frequency with which they reviewed *others'* commits, and at their reaction to reviews of their own commits. In the project's issue tracker, how often did the candidate respond constructively to incoming bug reports or contribute useful information to a bug ticket? Visit a threaded view of the project's discussion forums and see how often the candidate's posts were responded to, and what the general tone of the responses was. Someone who consistently causes negative reactions from others in the project may have social problems as a collaborator, which is important to know independently of the candidate's raw technical ability.

If candidate is applying for a position that would involve working on an open source project, but seems to have little or no open source experience themselves, this is not necessarily a showstopper, but it's a sign that you should ask some probing questions, and that you should expect some ramp-up time if you hire them. If the candidate is young and inexperienced in general, then lack of participation in open source is easy to understand. However, if the candidate has been a programmer for a while, and especially if they already have experience as a user of some of the open source software you'd be hiring them to work on, and yet they have never participated much in that project except to download and use it, then you should ask them questions about why. There is nothing wrong with being uninvolved as a participant in software that one uses. However, if you're hiring someone to *be* a participant in a project, and they already had a chance to be and chose not to, that implies a lack of intrinsic motivation to participate and may indicate that this person's temperament is not what you're looking for. Or there may be other reasons — for example, the candidate's prior management forbade them from participating. Whatever the reasons are, you should make sure you find out.

Evaluating Open Source Projects

Although this book is mainly about how to launch and run new open source projects, that topic is inextricably linked to the problem of evaluating existing open source projects. You can't know whether you need to start a new one until you've evaluated what's out there (as explained in the section called “But First, Look Around” in Chapter 2, *Getting Started*). Furthermore, even in a new project, you'll usually still be building on existing open source components, and will often be in the position of choosing be-

[9]Brian Fitzpatrick has written about the usefulness of having an open source résumé in two articles, *The Virtual Referral* (onlamp.com/pub/a/onlamp/2005/07/14/osdevelopers.html [http://www.onlamp.com/pub/a/onlamp/2005/07/14/osdevelopers.html]) and *The Virtual Internship* (onlamp.com/pub/a/onlamp/2005/08/01/opensourcedevelopers.html [http://www.onlamp.com/pub/a/onlamp/2005/08/01/opensourcedevelopers.html]).

tween different projects that implement the same basic functionality. That is not just a technical choice; it's also about the social health of those other projects: how large and diverse are their developer communities? Do they get new contributors on a regular basis? Do they handle incoming bug reports in a reasonable way? Do they make stable releases frequently enough for your needs? And so on.

Evaluating open source projects is certainly an art, not a science. However, there are some shortcuts that experienced people use. Below is what has worked for me, and by "worked", I mean that when I have applied these evaluation techniques to a project and then checked in with that project months or years later, I have generally found that its current circumstances are in line with what the evaluation predicted.

Look at bug tracker activity first.

> The most reliable signals of project health can usually be found in the bug tracker. Look at the rate of issue filings and the number of unique filers (because that's a proxy for the size and level of engagement of the user base), and look at how often project developers respond in bug tickets — and at *how* they respond: are they constructive? Do they interact well with both the reporter and with other developers? Is it always the same developer responding, or is responsiveness well-distributed throughout the development team? More bug reports is better, by the way, as discussed in the section called “Version Control and Bug Tracker Access” in Chapter 2, *Getting Started*. The rate at which bug reports are *closed* is not as important as you might think; in a healthy project with an active user base, bug reports will probably be filed faster than the development team can close them. The important question is not rate of resolution, but how the project responds to and organizes the influx of reports.

Measure commit diversity, not commit rate.

> Look at the distribution of commits across committers, not just at the raw frequency of commits. Does the project have a variety people working together in a sustained way? Too often, evaluators look just at the commit rate, but the rate isn't very informative — knowing the number of commits per week doesn't tell you anything except that (perhaps) someone keeps making typos and keeps correcting them in new commits. If have time to look at the content of individual commits, then look at how often one developer's commit is a response to (refers to) some other developer's previous commit. This tells you that group code review is going on, and the more of that you see, the better the project is doing.

Evaluate organizational diversity.

> In addition to looking for a variety of individual identities, see if you can tell how many different *organizations* are participating in the project — in particular, commercial organizations. If a number of different sources of money are all investing in a project, that's a sign that that project is going to be around for the long term. (See the discussion of "bus factor" in Chapter 2, *Getting Started*.)

Discussion forums.

> If the project has discussion forums, scan them quickly looking for signs of a functional community. Specifically, whenever you see a long thread, spot check responses from core developers coming late in the thread. Are they summarizing constructively, and taking steps to bring the thread to a decision while remaining polite? If you see a lot of flame wars going on, that's often a sign that energy is going into argument instead of into development.

News, announcements, and releases.

> Any project that is functioning well will usually have made announcements within the past few months. Check the project's front page, news feed, Twitter or other microblog accounts, etc. If things are quiet on stage, they're probably quiet backstage too.

This is only a quick introduction to the art of evaluating projects, but even using just the steps above can save you a lot of trouble. I have found them particularly useful when evaluating the two sides of a recent fork[10]. Even in a recent fork, it is often possible to tell, just by looking at some of the signals described above, which side will flourish over the long term.

Crowdfunding and Bounties

Perhaps unfairly, I will group crowdfunding campaigns and bounty-based development incentives together here, not because they are the same thing, but because to the extent that they are problematic as ways of funding free software development, their problems are similar.

Crowdfunding refers to many funders — often mostly individuals — coming together to fund a particular piece of development. Crowdfunding campaigns are usually either "all or nothing", meaning that each funder pledges money toward a total threshold and the pledges are collected only if the threshold is met, or "keep it all", which is essentially traditional donation: funds go immediately to the recipient whether or not a stated goal amount is ever met. Goteo.org [https://goteo.org/] and Kickstarter.com [https://Kickstarter.com/] are probably the best-known examples of all-or-nothing crowdfunding services, though there are many others (I like Goteo because their platform is itself free software, and because it is meant specifically for freely-licensed projects, whereas Kickstarter does not take a position on restrictiveness of licensing). There are also sites like Indiegogo [https://www.indiegogo.com/] that support both models[11]

Bounties are one-time rewards for completing specific tasks, such as fixing a bug or implementing a new feature. Bounties are often offered directly by the interested parties, since there is no need for a pledge-collecting system, but the site BountySource.com [https://www.bountysource.com/] also serves as a clearinghouse for open source development bounties.

While both crowdfunding and bounties have funded some open source work, they have not been a major economic force compared to salaried or contracted development. This does not mean you shouldn't consider them: depending on the problem you're trying to solve, and on the shapes of solutions you're willing to accept, crowdfunding or bounty funding might be a good answer. The problem they share is that they are structured around development as a *one-time event* rather than as an ongoing process. This would be problematic for any kind of software development, but is especially so for open source development, which if anything is is optimized more for low-intensity, long-term investment rather than for high-intensity burst investment. Both crowdfunding campaigns and bounty prizes are more compatible with high-intensity, one-time bursts of activity, and do not provide for ongoing maintenance or investment past the completion of the campaign goal or prize condition.[12]

A crowdfunding campaign can sometimes be a good way to get a project launched, but generally is not a way to fund development after the initial launch. Successive crowdfunding campaigns for later stages of development or for releases will inevitably tire out even a willing and supportive audience. There is a reason why long-running charities, for example the public radio network in the United States, seek to develop sustaining funders (euphemistically called "members" even though they often have no governance role) to provide a long-term, stable revenue stream, and then raise funds for specific one-time efforts separately from that.

If you do launch a crowdfunding campaign, take a close look at how other open source projects have run theirs. There are a number of useful techniques that can be learned from the successful ones. For exam-

[10] That is, a "social fork"; see "Social forks" versus "short forks". in Chapter 8, *Managing Participants*

[11] en.wikipedia.org/wiki/Comparison_of_crowdfunding_services [https://en.wikipedia.org/wiki/Comparison_of_crowdfunding_services].

[12] One service trying to solve that problem is Snowdrift.coop [https://snowdrift.coop/], which aims to provide sustainable funding for freely-licensed works using a carefully designed matching pledge model. Whether Snowdrift will succeed is unknowable as of this writing in mid-2015, since the service is still in a preliminary stage, but I am watching it with interest. Snowdrift also did a thorough survey, in the Fall of 2013, of funding platforms for free software, and posted their results at snowdrift.coop/p/snowdrift/w/en/othercrowdfunding [https://snowdrift.coop/p/snowdrift/w/en/othercrowdfunding]; it's worth a read if you're interested in this topic.

ple, most campaign sites have a mechanism for offering different rewards to backers at different monetary levels. You could offer a mention in a `SUPPORTERS` file in the project, and perhaps at higher levels a mention on a thank-you page on the project's web site. But more creatively — I first heard this idea from Michael Bernstein, and used it — you can offer to dedicate a commit to each backer at or above a certain level, by thanking the backer directly in the commit's log message. The nice thing about this is that it's decentralized and easy to administer: any developer on the project can help fulfill that reward. Individual developers can also offer free or discounted consulting about the project as a reward, though be careful not to sell too much of your time: the point of the campaign is to raise funds for development, not to turn the development team into a consulting team.

One thing that many crowdfunding campaigns do that I think is not appropriate for free software projects is to sell early access. That is, one of the rewards will be a "sneak preview" or "beta access" to in-progress versions, before the public release. The problem with this is that, for open source projects, the public is supposed to already have access to in-progress work. Access to an open source project should be limited by the time and interest of the parties *seeking* the access, not by the project. So learn what you can from other crowdfunding campaigns, but remember that some of the techniques used by most campaigns may not be suitable for an open source project that wants to keep the good will of its users and development community.

Finally, word of caution: if your project accepts donations, do some public planning of how the money will be used *before* it comes in. Discussions about how to allocate money tend to go a lot more smoothly when held before there's actual money to spend; and anyway, if there are significant disagreements, it's better to find that out when the money is still theoretical than when it's real.

Chapter 6. Communications

The ability to write clearly is perhaps the most important skill one can have in an open source environment. In the long run it matters more than programming talent. A great programmer with lousy communications skills can get only one thing done at a time, and even then may have trouble convincing others to pay attention. But a lousy programmer with good communications skills can coordinate and persuade many people to do many different things, and thereby have a significant effect on a project's direction and momentum.

There does not seem to be much correlation, in either direction, between the ability to write good code and the ability to communicate with one's fellow human beings. There is some correlation between programming well and describing technical issues well, but describing technical issues is only a tiny part of the communications in a project. Much more important is the ability to empathize with one's audience, to see one's own posts and comments as others see them, and to cause others to see their own posts with similar objectivity. Equally important is noticing when a given medium or communications method is no longer working well, perhaps because it doesn't scale as the number of users increases, and taking the time to do something about it.

All of which is obvious in theory—what makes it hard in practice is that free software development environments are bewilderingly diverse both in audiences and in communications mechanisms. Should a given thought be expressed in a post to the mailing list, as an annotation in the bug tracker, or as a comment in the code? When answering a question in a public forum, how much knowledge can you assume on the part of the reader, given that "the reader" is not only the one who asked the question in the first place, but all those who might see your response? How can the developers stay in constructive contact with the users, without getting swamped by feature requests, spurious bug reports, and general chatter? How do you tell when a medium has reached the limits of its capacity, and what do you do about it?

Solutions to these problems are usually partial, because any particular solution is eventually made obsolete by project growth or changes in project structure. They are also often *ad hoc*, because they're improvised responses to dynamic situations. All participants need to be aware of when and how communications can become bogged down, and be involved in solutions. Helping people do this is a big part of managing an open source project. The sections that follow discuss both how to conduct your own communications, and how to make maintenance of communications mechanisms a priority for everyone in the project.[1]

You Are What You Write

Consider this: the only thing anyone knows about you on the Internet comes from what you write, or what others write about you. You may be brilliant, perceptive, and charismatic in person—but if your emails are rambling and unstructured, people will assume that's the real you. Or perhaps you really are rambling and unstructured in person, but no one need ever know it, if your posts are lucid and informative.

Devoting some care to your writing will pay off hugely. Long-time free software hacker Jim Blandy tells the following story:

> Back in 1993, I was working for the Free Software Foundation, and we were beta-testing version 19 of GNU Emacs. We'd make a beta release every week or so, and peo-

[1]There has been some interesting academic research on this topic; for example, see *Group Awareness in Distributed Software Development* by Gutwin, Penner, and Schneider. This paper was online for a while, then unavailable, then online again at st.cs.uni-sb.de/edu/empirical-se/2006/PDFs/gutwin04.pdf [http://www.st.cs.uni-sb.de/edu/empirical-se/2006/PDFs/gutwin04.pdf]. So try there first, but be prepared to use a search engine if it moves again.

> ple would try it out and send us bug reports. There was this one guy whom none of us had met in person but who did great work: his bug reports were always clear and led us straight to the problem, and when he provided a fix himself, it was almost always right. He was top-notch.
>
> Now, before the FSF can use code written by someone else, we have them do some legal paperwork to assign their copyright interest to that code to the FSF. Just taking code from complete strangers and dropping it in is a recipe for legal disaster.
>
> So I emailed the guy the forms, saying, "Here's some paperwork we need, here's what it means, you sign this one, have your employer sign that one, and then we can start putting in your fixes. Thanks very much."
>
> He sent me back a message saying, "I don't have an employer."
>
> So I said, "Okay, that's fine, just have your university sign it and send it back."
>
> After a bit, he wrote me back again, and said, "Well, actually... I'm thirteen years old and I live with my parents."

Because that kid didn't write like a thirteen-year-old, no one knew that's what he was. Following are some ways to make your writing give a good impression too.

Structure and Formatting

Don't fall into the trap of writing everything as though it were a cell phone text message. Write in complete sentences, capitalizing the first word of each sentence, and use paragraph breaks where needed. This is most important in emails and other composed writings. In IRC or similarly ephemeral forums, it's generally okay to leave out capitalization, use compressed forms of common expressions, etc. Just don't carry those habits over into more formal, persistent forums. Emails, documentation, bug reports, and other pieces of writing that are intended to have a permanent life should be written using standard grammar and spelling, and have a coherent narrative structure. This is not because there's anything inherently good about following arbitrary rules, but rather that these rules are *not* arbitrary: they evolved into their present forms because they make text more readable, and you should adhere to them for that reason. Readability is desirable not only because it means more people will understand what you write, but because it makes you look like the sort of person who takes the time to communicate clearly: that is, someone worth paying attention to.

For email in particular, experienced open source developers have settled on certain conventions:

Send plain text mails only, not HTML, RichText, or other formats that might get mangled by certain online archives or text-based mail readers. When including screen output, snippets of code, or other preformatted text, offset it clearly, so that even a lazy eye can easily see the boundaries between your prose and the material you're quoting. If the overall structure of your post is still visible from five meters away, you're doing it right.

For preformatted blocks, such as quoted code or error messages, try to stay under 80 columns wide, which has become the *de facto* standard terminal width (that is, some people may use wider displays, but no one uses a narrower one). By making your lines a little *less* than 80 columns, you leave room for a few levels of quoting characters to be added in others' replies without forcing a rewrapping of your preformatted text.

When quoting someone else's mail, insert your responses where they're most appropriate, at several different places if necessary, and trim off the parts of their mail you didn't use. If you're writing a quick response that applies to their entire post, and your response will be sensible even to someone who hasn't

read the original, then it's okay to *top-post* (that is, to put your response above the quoted text of their mail); otherwise, quote the relevant portion of the original text first, followed by your response.

Construct the subject lines of new mails carefully. It's the most important line in your mail, because it allows each other person in the project to decide whether or not to read more. Modern mail reading software organizes groups of related messages into threads, which can be defined not only by a common subject, but by various other headers (which are sometimes not displayed). It follows that if a thread starts to drift to a new topic, you can—and should—adjust the subject line accordingly when replying. The thread's integrity will be preserved, due to those other headers, but the new subject will help people looking at an overview of the thread know that the topic has drifted. Likewise, if you really want to start a new topic, do it by posting a fresh mail, not by replying to an existing mail and changing the subject. Otherwise, your mail would still be grouped in to the same thread as what you're replying to, and thus fool people into thinking it's about something it's not. Again, the penalty would not only be the waste of their time, but the slight dent in your credibility as someone fluent in using communications tools.

Content

Well-formatted mails attract readers, but content keeps them. No set of fixed rules can guarantee good content, of course, but there are some principles that make it more likely.

Make things easy for your readers. There's a ton of information floating around in any active open source project, and readers cannot be expected to be familiar with most of it—indeed, they cannot always be expected to know how to become familiar. Wherever possible, your posts should provide information in the form most convenient for readers. If you have to spend an extra two minutes to dig up the URL to a particular thread in the mailing list archives, in order to save your readers the trouble of doing so, it's worth it. If you have to spend an extra 5 or 10 minutes summarizing the conclusions so far of a complex thread, in order to give people context in which to understand your post, then do so. Think of it this way: the more successful a project, the higher the reader-to-writer ratio in any given forum. If every post you make is seen by *n* people, then as *n* rises, the worthwhileness of expending extra effort to save those people time rises with it. And as people see you imposing this standard on yourself, they will work to match it in their own communications. The result is, ideally, an increase in the global efficiency of the project: when there is a choice between *n* people making an effort and one person doing so, the project prefers the latter.

Don't engage in hyperbole. Exaggerating in online posts is a classic arms race. For example, a person reporting a bug may worry that the developers will not pay sufficient attention, so he'll describe it as a severe, showstopper problem that is preventing him (and all his friends/coworkers/cousins) from using the software productively, when it's actually only a mild annoyance. But exaggeration is not limited to users—programmers often do the same thing during technical debates, particularly when the disagreement is over a matter of taste rather than correctness:

> "Doing it that way would make the code totally unreadable. It'd be a maintenance nightmare, compared to J. Random's proposal..."

The same sentiment actually becomes *stronger* when phrased less sharply:

> "That works, but it's less than ideal in terms of readability and maintainability, I think. J. Random's proposal avoids those problems because it..."

You will not be able to rid the project of hyperbole completely, and in general it's not necessary to do so. Compared to other forms of miscommunication, hyperbole is not globally damaging—it hurts mainly the perpetrator. The recipients can compensate, it's just that the sender loses a little more credibility each time. Therefore, for the sake of your own influence in the project, try to err on the side of moderation. That way, when you *do* need to make a strong point, people will take you seriously.

Edit twice. For any message longer than a medium-sized paragraph, reread it from top to bottom before sending it but after you think it's done the first time. This is familiar advice to anyone who's taken a composition class, but it's especially important in online discussion. Because the process of online composition tends to be highly discontinuous (in the course of writing a message, you may need to go back and check other mails, visit certain web pages, run a command to capture its debugging output, etc.), it's especially easy to lose your sense of narrative place. Messages that were composed discontinuously and not checked before being sent are often recognizable as such, much to the chagrin (or so one would hope) of their authors. Take the time to review what you send. The more your posts hold together structurally, the more they will be read by others.

Tone

After writing thousands of messages, you will probably find your style tending toward the terse. This seems to be the norm in most technical forums, and there's nothing wrong with it per se. A degree of terseness that would be unacceptable in normal social interactions is simply the default for free software hackers. Here's a response I once drew on a mailing list about some free content management software, quoted in full:

```
Can you possibly elaborate a bit more on exactly what problems
you ran into, etc?

Also:

What version of Slash are you using? I couldn't tell from your
original message.

Exactly how did you build the apache/mod_perl source?

Did you try the Apache 2.0 patch that was posted about on
slashcode.com?

  Shane
```

Now *that's* terse! No greeting, no sign-off other than his name, and the message itself is just a series of questions phrased as compactly as possible. His one declarative sentence was an implicit criticism of my original message. And yet, I was happy to see Shane's mail, and didn't take his terseness as a sign of anything other than him being a busy person. The mere fact that he was asking questions, instead of ignoring my post, meant that he was willing to spend some time on my problem.

Will all readers react positively to this style? Not necessarily; it depends on the person and the context. For example, if someone has just posted acknowledging that he made a mistake (perhaps he wrote a bug), and you know from past experience that this person tends to be a bit insecure, then while you may still write a compact response, you should make sure to leaven it with some sort of acknowledgment of his feelings. The bulk of your response might be a brief, engineer's-eye analysis of the situation, as terse as you want. But at the end, sign off with something indicating that your terseness is not to be taken as coldness. For example, if you've just given reams of advice about exactly how the person should fix the bug, then sign off with "Good luck, <your name here>" to indicate that you wish him well and are not mad. A strategically placed smiley face or other emoticlue can often be enough to reassure an interlocutor, too.

It may seem odd to focus as much on the participant's feelings as on the surface of what they say, but, to put it baldly, feelings affect productivity. Feelings are important for other reasons too, but even confining ourselves to purely utilitarian grounds, we may note that unhappy people write worse software

and tackle fewer bugs. Given the restricted nature of most electronic media, though, there will often be no overt clue about how a person is feeling. You will have to make an educated guess based on a) how most people would feel in that situation, and b) what you know of this particular person from past interactions. Some people prefer a more hands-off attitude, and simply deal with everyone at face value, the idea being that if a participant doesn't say outright that he feels a particular way, then one has no business treating him as though he does. I don't buy this approach, for a couple of reasons. One, people don't behave that way in real life, so why would they online? Two, since most interactions take place in public forums, people tend to be even more restrained in expressing emotions than they might be in private. To be more precise, they are often willing to express emotions directed at others, such as gratitude or outrage, but not emotions directed inwardly, such as insecurity or pride. Yet most humans work better when they know that others are aware of their state of mind. By paying attention to small clues, you can usually guess right most of the time, and motivate people to stay involved to a greater degree than they otherwise might.

I don't mean, of course, that your role is to be a group therapist, constantly helping everyone to get in touch with their feelings. But by paying careful attention to long-term patterns in people's behavior, you will begin to get a sense of them as individuals even if you never meet them face-to-face. And by being sensitive to the tone of your own writing, you can have a surprising amount of influence over how others feel, to the ultimate benefit of the project.

Recognizing Rudeness

One of the defining characteristics of open source culture is its distinctive notions of what does and does not constitute rudeness. While the conventions described below are not unique to free software development, nor even to software in general—they would be familiar to anyone working in mathematics, the hard sciences, or engineering disciplines—free software, with its porous boundaries and constant influx of newcomers, is an environment where these conventions are especially likely to be encountered by people unfamiliar with them. (This is one reason why it's good to be generous when trying to figure out whether someone has violated the code of conduct, in a project that has one — see the section called “Codes of Conduct” in Chapter 2, *Getting Started*.)

Let's start with the things that are *not* rude:

Technical criticism, even when direct and unpadded, is not rude. Indeed, it can be a form of flattery: the critic is saying, by implication, that the target is worth taking seriously, and is worth spending some time on. That is, the more viable it would have been to simply ignore someone's post, the more of a compliment it becomes to take the time to criticize it (unless the critique descends into an *ad hominem* attack or some other form of obvious rudeness, of course).

Blunt, unadorned questions, such as Shane's questions to me in the previously quoted email, are not rude either. Questions that in other contexts might seem cold, rhetorical, or even mocking, are often intended seriously, and have no hidden agenda other than eliciting information as quickly as possible. The famous technical support question "Is your computer plugged in?" is a classic example of this. The support person really does need to know if your computer is plugged in, and after the first few days on the job, has gotten tired of prefixing her question with polite blandishments ("I beg your pardon, I just want to ask a few simple questions to rule out some possibilities. Some of these might seem pretty basic, but bear with me..."). At this point, she doesn't bother with the padding anymore, she just asks straight out: is it plugged in or not? Equivalent questions are asked all the time on free software mailing lists. The intent is not to insult the recipient, but to quickly rule out the most obvious (and perhaps most common) explanations. Recipients who understand this and react accordingly win points for taking a broad-minded view without prompting. But recipients who react badly must not be reprimanded, either. It's just a collision of cultures, not anyone's fault. Explain amiably that your question (or criticism) had no hidden meanings; it was just meant to get (or transmit) information as efficiently as possible, nothing more.

So what *is* rude?

By the same principle under which detailed technical criticism is a form of flattery, failure to provide quality criticism can be a kind of insult. I don't mean simply ignoring someone's work, be it a proposal, code change, new ticket filing, or whatever. Unless you explicitly promised a detailed reaction in advance, it's usually okay to simply not react at all. People will assume you just didn't have time to say anything. But if you *do* react, don't skimp: take the time to really analyze things, provide concrete examples where appropriate, dig around in the archives to find related posts from the past, etc. Or if you don't have time to put in that kind of effort, but still need to write some sort of brief response, then state the shortcoming openly in your message ("I think there's a ticket filed for this, but unfortunately didn't have time to search for it, sorry"). The main thing is to recognize the existence of the cultural norm, either by fulfilling it or by openly acknowledging that one has fallen short this time. Either way, the norm is strengthened. But failing to meet that norm, while at the same time not explaining *why* you failed to meet it, is like saying the topic (and those participating in it) was not worth much of your time—that your time is more valuable than theirs. Better to show that your time is valuable by being terse than by being lazy.

There are many other forms of rudeness, of course, but most of them are not specific to free software development, and common sense is a good enough guide to avoid them. See also the section called “Nip Rudeness in the Bud” in Chapter 2, *Getting Started*, if you haven't already.

Face

There is a region in the human brain devoted specifically to recognizing faces. It is known informally as the "fusiform face area" and apparently its capabilities are at least partly inborn, not learned. It turns out that recognizing individual people is such a crucial survival skill that we have evolved specialized hardware to do it.

Internet-based collaboration is therefore psychologically odd, because it involves tight cooperation between human beings who almost never get to identify each other by the most natural, intuitive methods: facial recognition first of all, but also sound of voice, posture, etc. To compensate for this, try to use a consistent *screen name* everywhere. It should be the front part of your email address (the part before the @-sign), your IRC username, your repository committer name, your ticket tracker username, and so on. This name is your online "face": a short identifying string that serves some of the same purpose as your real face, although it does not, unfortunately, stimulate the same built-in hardware in the brain.

The screen name should be some intuitive permutation of your real name (mine, for example, is "kfogel"). In some situations it will be accompanied by your full name anyway, for example in mail headers:

```
From: "Karl Fogel" <kfogel@whateverdomain.com>
```

Actually, there are two things going on in that example. As mentioned earlier, the screen name matches the real name in an intuitive way. But also, the real name is *real*. That is, it's not some made-up appellation like:

```
From: "Wonder Hacker" <wonderhacker@whateverdomain.com>
```

There's a famous cartoon by Paul Steiner, from the July 5, 1993 issue of *The New Yorker*, that shows one dog logged into a computer terminal, looking down and telling another conspiratorially: "On the Internet, nobody knows you're a dog." This kind of thought probably lies behind a lot of the self-aggrandizing, meant-to-be-hip online identities people give themselves—as if calling oneself "Wonder Hacker" will actually cause people to believe one *is* a wonderful hacker. But the fact remains: even if no one knows you're a dog, you're still a dog. A fantastical online identity never impresses readers. Instead, it makes them think you're more into image than substance, or that you're simply insecure. Use your real

name for all interactions, or if for some reason you require anonymity, then make up a name that sounds like a perfectly normal real name, and use it consistently.

In addition to keeping your online face consistent, there are some things you can do to make it more attractive. If you have an official title (e.g., "doctor", "professor", "director"), don't flaunt it, nor even mention it except when it's directly relevant to the conversation. Hackerdom in general, and free software culture in particular, tends to view title displays as exclusionary and a sign of insecurity. It's okay if your title appears as part of a standard signature block at the end of every mail you send, just don't ever use it as a tool to bolster your position in a discussion—the attempt is guaranteed to backfire. You want folks to respect the person, not the title.

Speaking of signature blocks: keep them small and tasteful, or better yet, nonexistent. Avoid large legal disclaimers tacked on to the end of every mail, especially when they express sentiments incompatible with participation in a free software project. For example, the following classic of the genre appears at the end of every post a particular user makes to a certain project mailing list:

```
IMPORTANT NOTICE

If you have received this e-mail in error or wish to read our e-mail
disclaimer statement and monitoring policy, please refer to the
statement below or contact the sender.

This communication is from Deloitte & Touche LLP.  Deloitte &
Touche LLP is a limited liability partnership registered in England
and Wales with registered number OC303675.  A list of members' names
is available for inspection at Stonecutter Court, 1 Stonecutter
Street, London EC4A 4TR, United Kingdom, the firm's principal place of
business and registered office.  Deloitte & Touche LLP is
authorised and regulated by the Financial Services Authority.

This communication and any attachments contain information which is
confidential and may also be privileged.  It is for the exclusive use
of the intended recipient(s).  If you are not the intended
recipient(s) please note that any form of disclosure, distribution,
copying or use of this communication or the information in it or in
any attachments is strictly prohibited and may be unlawful.  If you
have received this communication in error, please return it with the
title "received in error" to IT.SECURITY.UK@deloitte.co.uk then delete
the email and destroy any copies of it.

E-mail communications cannot be guaranteed to be secure or error free,
as information could be intercepted, corrupted, amended, lost,
destroyed, arrive late or incomplete, or contain viruses.  We do not
accept liability for any such matters or their consequences.  Anyone
who communicates with us by e-mail is taken to accept the risks in
doing so.

When addressed to our clients, any opinions or advice contained in
this e-mail and any attachments are subject to the terms and
conditions expressed in the governing Deloitte & Touche LLP client
engagement letter.

Opinions, conclusions and other information in this e-mail and any
```

```
attachments which do not relate to the official business of the firm
are neither given nor endorsed by it.
```

For someone who's just showing up to ask a question now and then, that huge disclaimer looks a bit silly but probably doesn't do any lasting harm. However, if this person wanted to participate actively in the project, that legal boilerplate would start to have a more insidious effect. It would send at least two potentially destructive signals: first, that this person doesn't have full control over his tools—he's trapped inside some corporate mailer that tacks an annoying message to the end of every email, and he hasn't got any way to route around it—and second, that he has little or no organizational support for his free software activities. True, the organization has clearly not banned him outright from posting to public lists, but it has made his posts look distinctly unwelcoming, as though the risk of letting out confidential information must trump all other priorities.

If you work for an organization that insists on adding such signature blocks to all outgoing mail, and you can't get the policy changed, then consider using your personal email account to post, even if you're being paid by your employer for your participation in the project.

Avoiding Common Pitfalls

Don't Post Without a Purpose

A common pitfall in online project participation is to think that you have to respond to everything. You don't. First of all, there will usually be more threads going on than you can keep track of, at least after the project really gets going. Second, even in the threads that you have decided to engage in, much of what people say will not require a response. Development forums in particular tend to be dominated by three kinds of messages:

1. Messages proposing something non-trivial
2. Messages expressing support or opposition to something someone else has said
3. Summing-up messages

None of these *inherently* requires a response, particularly if you can be fairly sure, based on watching the thread so far, that someone else is likely to say what you would have said anyway. (If you're worried that you'll be caught in a wait-wait loop because all the others are using this tactic too, don't be; there's almost always *someone* out there who'll feel like jumping into the fray.) A response should be motivated by a definite purpose. Ask yourself first: do you know what you want to accomplish? And second: will it not get accomplished unless you say something?

Two good reasons to add your voice to a thread are a) when you see a flaw in a proposal and suspect that you're the only one who sees it, and b) when you see that miscommunication is happening between others, and know that you can fix it with a clarifying post. It's also generally fine to post just to thank someone for doing something, or to say "Me too!", because a reader can tell right away that such posts do not require any response or further action, and therefore the mental effort demanded by the post ends cleanly when the reader reaches the last line of the mail. But even then, think twice before saying something; it's always better to leave people wishing you'd post more than wishing you'd post less. (The second half of Poul-Henning Kamp's "bikeshed" post, referenced from the section called “The Smaller the Topic, the Longer the Debate”, offers some further thoughts about how to behave on a busy mailing list.)

Productive vs Unproductive Threads

On a busy mailing list, you have two imperatives. One, obviously, is to figure out what you need to pay attention to and what you can ignore. The other is to behave in a way that avoids *causing* noise: not only

do you want your own posts to have a high signal/noise ratio, you also want them to be the sorts of messages that stimulate *other* people to either post with a similarly high signal/noise ratio, or not post at all.

To see how to do that, let's consider the context in which it is done. What are some of the hallmarks of an unproductive thread?

- Arguments that have been made already start to be repeated in the same thread, as though the poster thinks no one heard them the first time.
- Increasing levels of hyperbole and involvement as the stakes get smaller and smaller.
- A majority of comments coming from people who do little or nothing, while the people who tend to get things done are silent.
- Many ideas discussed without clear proposals ever being made. (Of course, any interesting idea starts out as an imprecise vision; the important question is what direction it goes from there. Does the thread seem to be turning the vision into something more concrete, or is it spinning off into sub-visions, side-visions, and ontological disputes?)

Just because a thread is not productive at first doesn't mean it's a waste of time. It might be about an important topic, in which case the fact that it's not making any headway is all the more troublesome.

Guiding a thread toward usefulness without being pushy is an art. It won't work to simply admonish people to stop wasting their time, or to ask them not to post unless they have something constructive to say. You may, of course, think these things privately, but if you say them out loud then you will be offensive—and ineffective. Instead, you have to suggest conditions for further progress: give people a route, a path to follow that leads to the results you want, yet without sounding like you're dictating conduct. The distinction is largely one of tone. For example, this is bad:

> *This discussion is going nowhere. Can we please drop this topic until someone has a patch to implement one of these proposals? There's no reason to keep going around and around saying the same things. Code speaks louder than words, folks.*

Whereas this is good:

> *Several proposals have been floated in this thread, but none have had all the details fleshed out, at least not enough for an up-or-down vote. Yet we're also not saying anything new now; we're just reiterating what has been said before. So the best thing at this point would probably be for further posts to contain either a complete specification for the proposed behavior, or a patch. Then at least we'd have a definite action to take (i.e., get consensus on the specification, or apply and test the patch).*

Contrast the second approach with the first. The second way does not draw a line between you and the others, or accuse them of taking the discussion into a spiral. It talks about "we", which is important whether or not you actually participated in the thread before now, because it reminds everyone that even those who have been silent thus far still have a stake in the thread's outcome. It describes why the thread is going nowhere, but does so without pejoratives or judgements—it just dispassionately states some facts. Most importantly, it offers a positive course of action, so that instead of people feeling like discussion is being closed off (a restriction against which they can only be tempted to rebel), they will feel as if they're being offered a way to take the conversation to a more constructive level, if they're willing to make the effort. This is a standard the most productive people will naturally want to meet.

You won't always want a thread to make it to the next level of constructiveness—sometimes you'll want it to just go away. The purpose of your post, then, is to make it do one or the other. If you can tell from the way the thread has gone so far that no one is actually *going* to take the steps you suggested, then

your post effectively shuts down the thread without seeming to do so. Of course, there isn't any foolproof way to shut down a thread, and even if there were, you wouldn't want to use it. But asking participants to either make visible progress or stop posting is perfectly defensible, if done diplomatically. Be wary of quashing threads prematurely, however. Some amount of speculative chatter can be productive, depending on the topic, and asking for it to be resolved too quickly will stifle the creative process, as well as make you look impatient.

Don't expect any thread to stop on a dime. There will probably still be a few posts after yours, either because mails got crossed in the pipe, or because people want to have the last word. This is nothing to worry about, and you don't need to post again. Just let the thread peter out, or not peter out, as the case may be. You can't have complete control; on the other hand, you can expect to have a statistically significant effect across many threads.

The Smaller the Topic, the Longer the Debate

Although discussion can meander in any topic, the probability of meandering goes up as the technical difficulty of the topic goes down. After all, the greater the technical complexity, the fewer participants can really follow what's going on. Those who can are likely to be the most experienced developers, who have already taken part in such discussions many times before, and know what sort of behavior is likely to lead to a consensus everyone can live with.

Thus, consensus is hardest to achieve in technical questions that are simple to understand and easy to have an opinion about, and in "soft" topics such as organization, publicity, funding, etc. People can participate in those arguments forever, because there are no qualifications necessary for doing so, no clear ways to decide (even afterward) if a decision was right or wrong, and because simply outwaiting other discussants is sometimes a successful tactic.

The principle that the amount of discussion is inversely proportional to the complexity of the topic has been around for a long time, and is known informally as the *Bikeshed Effect*. Here is Poul-Henning Kamp's explanation of it, from a now-famous post made to BSD developers:

> It's a long story, or rather it's an old story, but it is quite short actually. C. Northcote Parkinson wrote a book in the early 1960'ies, called "Parkinson's Law", which contains a lot of insight into the dynamics of management.
>
> [...]
>
> In the specific example involving the bike shed, the other vital component is an atomic power-plant, I guess that illustrates the age of the book.
>
> Parkinson shows how you can go in to the board of directors and get approval for building a multi-million or even billion dollar atomic power plant, but if you want to build a bike shed you will be tangled up in endless discussions.
>
> Parkinson explains that this is because an atomic plant is so vast, so expensive and so complicated that people cannot grasp it, and rather than try, they fall back on the assumption that somebody else checked all the details before it got this far. Richard P. Feynmann gives a couple of interesting, and very much to the point, examples relating to Los Alamos in his books.
>
> A bike shed on the other hand. Anyone can build one of those over a weekend, and still have time to watch the game on TV. So no matter how well prepared, no matter how reasonable you are with your proposal, somebody will seize the chance to show that he is doing his job, that he is paying attention, that he is *here*.

> In Denmark we call it "setting your fingerprint". It is about personal pride and prestige, it is about being able to point somewhere and say "There! *I* did that." It is a strong trait in politicians, but present in most people given the chance. Just think about footsteps in wet cement.

(Kamp's complete post is very much worth reading, too; see bikeshed.com [http://bikeshed.com/].)

Anyone who's ever taken regular part in group decision-making will recognize what Kamp is talking about. However, it is usually impossible to persuade *everyone* to avoid painting bikesheds. The best you can do is point out that the phenomenon exists, when you see it happening, and persuade the senior developers—the people whose posts carry the most weight—to drop their paintbrushes early, so at least they're not contributing to the noise. Bikeshed painting parties will never go away entirely, but you can make them shorter and less frequent by spreading an awareness of the phenomenon in the project's culture.

Avoid Holy Wars

A *holy war* is a dispute, often but not always over a relatively minor issue, which is not resolvable on the merits of the arguments, but where people feel passionate enough to continue arguing anyway in the hope that their side will prevail. Holy wars are not quite the same as bikeshed painting. People painting bikesheds may be quick to jump in with an opinion, but they won't necessarily feel strongly about it, and indeed will sometimes express other, incompatible opinions, to show that they understand all sides of the issue. In a holy war, on the other hand, understanding the other sides is a sign of weakness. In a holy war, everyone knows there is One Right Answer; they just don't agree on what it is.

Once a holy war has started, it generally cannot be resolved to everyone's satisfaction. It does no good to point out, in the midst of a holy war, that a holy war is going on. Everyone knows that already. Unfortunately, a common feature of holy wars is disagreement on the very question of *whether* the dispute is resolvable by continued discussion. Viewed from outside, it is clear that neither side is changing the other's mind. Viewed from inside, the other side is being obtuse and not thinking clearly, but they might come around if browbeaten enough. Now, I am *not* saying there's never a right side in a holy war. Sometimes there is—in the holy wars I've participated in, it's always been my side, of course. But it doesn't matter, because there's no algorithm for convincingly demonstrating that one side or the other is right.

A common, but unsatisfactory, way people try to resolve holy wars is to say "We've already spent far more time and energy discussing this than it's worth! Can we please just drop it?" There are two problems with this. First, that time and energy has already been spent and can never be recovered—the only question now is, how much *more* effort remains? If some people feel that just a little more discussion will resolve the issue in their favor, then it still makes sense (from their point of view) to continue.

The other problem with asking for the matter to be dropped is that this is often equivalent to allowing one side, the status quo, to declare victory by inaction. And in some cases, the status quo is known to be unacceptable anyway: everyone agrees that some decision must be made, some action taken. Dropping the subject would be worse for everyone than simply giving up the argument would be for anyone. But since that dilemma applies to all equally, it's still possible to end up arguing forever about what to do.

So how should you handle holy wars?

The first answer is, try to set things up so they don't happen. This is not as hopeless as it sounds:

You can anticipate certain standard holy wars: they tend to come up over programming languages, licenses (see the section called “The GPL and License Compatibility” in Chapter 9, *Legal Matters: Licenses, Copyrights, Trademarks and Patents*), reply-to munging (see the section called “The Great Reply-to Debate” in Chapter 3, *Technical Infrastructure*), and a few other topics. Each project usually has a holy war or two all its own, as well, which longtime developers will quickly become familiar with.

The techniques for stopping holy wars, or at least limiting their damage, are pretty much the same everywhere. Even if you are positive your side is right, try to find *some* way to express sympathy and understanding for the points the other side is making. Often the problem in a holy war is that because each side has built its walls as high as possible and made it clear that any other opinion is sheer foolishness, the act of surrendering or changing one's mind becomes psychologically unbearable: it would be an admission not just of being wrong, but of having been *certain* and still being wrong. The way you can make this admission palatable for the other side is to express some uncertainty yourself—precisely by showing that you understand the arguments they are making and find them at least sensible, if not finally persuasive. Make a gesture that provides space for a reciprocal gesture, and usually the situation will improve. You are no more or less likely to get the technical result you wanted, but at least you can avoid unnecessary collateral damage to the project's morale.

When a holy war can't be avoided, decide early how much you care, and then be willing to publicly give up. When you do so, you can say that you're backing out because the holy war isn't worth it, but don't express any bitterness and *don't* take the opportunity for a last parting shot at the opposing side's arguments. Giving up is effective only when done gracefully.

Programming language holy wars are a bit of a special case, because they are often highly technical, yet many people feel qualified to take part in them, and the stakes are very high, since the result may determine what language a good portion of the project's code is written in. The best solution is to choose the language early, with buy-in from influential initial developers, and then defend it on the grounds that it's what you are all comfortable writing in, *not* on the grounds that it's better than some other language that could have been used instead. Never let the conversation degenerate into an academic comparison of programming languages (this seems to happen especially often when someone brings up Perl, for some reason); that's a death topic that you must simply refuse to be drawn into.

For more historical background on holy wars, see catb.org/~esr/jargon/html/H/holy-wars.html [http://catb.org/~esr/jargon/html/H/holy-wars.html], and the paper by Danny Cohen that popularized the term, ietf.org/rfc/ien/ien137.txt [https://www.ietf.org/rfc/ien/ien137.txt].

The "Noisy Minority" Effect

In any mailing list discussion, it's easy for a small minority to give the impression that there is a great deal of dissent, by flooding the list with numerous lengthy emails. It's a bit like a filibuster, except that the illusion of widespread dissent is even more powerful, because it's divided across an arbitrary number of discrete posts and most people won't bother to keep track of who said what, when. They'll just have an instinctive impression that the topic is very controversial, and wait for the fuss to die down.

The best way to counteract this effect is to point it out very clearly and provide supporting evidence showing how small the actual number of dissenters is, compared to those in agreement. In order to increase the disparity, you may want to privately poll people who have been mostly silent, but who you suspect would agree with the majority. Don't say anything that suggests the dissenters were deliberately trying to inflate the impression they were making. Chances are they weren't, and even if they were, there would be no strategic advantage to pointing it out. All you need do is show the actual numbers in a side-by-side comparison, and people will realize that their intuition of the situation does not match reality.

This advice doesn't just apply to issues with clear for-and-against positions. It applies to any discussion where a fuss is being made but it's not clear that most people consider the issue under discussion to be a real problem. After a while, if you agree that the issue is not worthy of action, and can see that it has failed to get much traction (even if it has generated a lot of mails), you can just observe publicly that it's not getting traction. If the "Noisy Minority" effect has been at work, your post will seem like a breath of fresh air. Most people's impression of the discussion up to that point will have been somewhat murky: "Huh, it sure feels like there's some big deal here, because there sure are a lot of posts, but I can't see any clear progress happening." By explaining how the form of the discussion made it appear more turbulent

than it really was, you retrospectively give it a new shape, through which people can recast their understanding of what transpired.

Difficult People

Difficult people are no easier to deal with in electronic forums than they are in person. By "difficult" I don't mean "rude". Rude people are annoying, but they're not necessarily difficult. This book has already discussed how to handle them: comment on the rudeness the first time, and from then on, either ignore them or treat them the same as anyone else. If they continue being rude, they will usually make themselves so unpopular as to have no influence on others in the project, so they are a self-containing problem.

The really difficult cases are people who are not overtly rude, but who manipulate or abuse the project's processes in a way that ends up costing other people time and energy, yet do not bring any benefit to the project[2].

Often, such people look for wedgepoints in the project's procedures, to give themselves more influence than they might otherwise have. This is much more insidious than mere rudeness, because neither the behavior nor the damage it causes is apparent to casual observers. A classic example is the filibuster, in which someone (always sounding as reasonable as possible, of course) keeps claiming that the matter under discussion is not ready for resolution, and offers more and more possible solutions, or new viewpoints on old solutions, when what is really going on is that he senses that a consensus or a ballot is about to form and he doesn't like where it's headed. Another example is when there's a debate that won't converge on consensus, but the group tries to at least clarify the points of disagreement and produce a summary for everyone to refer to from then on. The obstructionist, who knows the summary may lead to a result he doesn't like, will often try to delay even the summary, by relentlessly complicating the question of what should be in it, either by objecting to reasonable suggestions or by introducing unexpected new items.

Handling Difficult People

To counteract such behavior, it helps to understand the mentality of those who engage in it. People generally do not do it consciously. No one wakes up in the morning and says to himself: "Today I'm going to cynically manipulate procedural forms in order to be an irritating obstructionist." Instead, such actions are often preceded by a semi-paranoid feeling of being shut out of group interactions and decisions. The person feels he is not being taken seriously, or (in the more severe cases) that there is almost a conspiracy against him—that the other project members have decided to form an exclusive club, of which he is not a member. This then justifies, in his mind, taking rules literally and engaging in a formal manipulation of the project's procedures, in order to *make* everyone else take him seriously. In extreme cases, the person can even believe that he is fighting a lonely battle to save the project from itself.

It is the nature of such an attack from within that not everyone will notice it at the same time, and some people may not see it at all unless presented with very strong evidence. This means that neutralizing it can be quite a bit of work. It's not enough to persuade yourself that it's happening; you have to marshal enough evidence to persuade others too, and then you have to distribute that evidence in a thoughtful way.

Given that it's so much work to fight, it's often better just to tolerate it for a while. Think of it like a parasitic but mild disease: if it's not too debilitating, the project can afford to remain infected, and medicine

[2]For an extended discussion of one particular subspecies of difficult person, see Amy Hoy's hilariously on-target *Help Vampires: A Spotter's Guide [http://slash7.com/2006/12/22/vampires/]*. Quoting Hoy: "It's so regular you could set your watch by it. The decay of a community is just as predictable as the decay of certain stable nuclear isotopes. As soon as an open source project, language, or what-have-you achieves a certain notoriety — its half-life, if you will — *they* swarm in, seemingly draining the very life out of the community itself. *They* are the Help Vampires. And I'm here to stop them..."

might have harmful side effects. However, if it gets too damaging to tolerate, then it's time for action. Start gathering notes on the patterns you see. Make sure to include references to public archives—this is one of the reasons the project keeps records, so you might as well use them. Once you've got a good case built, start having private conversations with other project participants. Don't tell them what you've observed; instead, first ask them what they've observed. This may be your last chance to get unfiltered feedback about how others see the troublemaker's behavior; once you start openly talking about it, opinion will become polarized and no one will be able to remember what he formerly thought about the matter.

If private discussions indicate that at least some others see the problem too, then it's time to do something. That's when you have to get *really* cautious, because it's very easy for this sort of person to try to make it appear as though you're picking on them unfairly. Whatever you do, never accuse them of maliciously abusing the project's procedures, of being paranoid, or, in general, of any of the other things that you suspect are probably true. Your strategy should be to look both more reasonable and more concerned with the overall welfare of the project, with the goal of either reforming the person's behavior, or getting them to go away permanently. Depending on the other developers, and your relationship with them, it may be advantageous to gather allies privately first. Or it may not; that might just create ill will behind the scenes, if people think you're engaging in an improper whispering campaign.

Remember that although the other person may be the one behaving destructively, *you* will be the one who appears destructive if you make a public charge that you can't back up. Be sure to have plenty of examples to demonstrate what you're saying, and say it as gently as possible while still being direct. You may not persuade the person in question, but that's okay as long as you persuade everyone else.

Case study

I remember only a few situations, in more than 20 years of working in free software, where things got so bad that we actually had to ask someone to stop posting altogether. In the example I'll use here, the person was not rude, and sincerely wanted only to be helpful. He just didn't know when to post and when not to post. Our lists were open to the public, and he was posting so often, and asking questions on so many different topics, that it was getting to be a noise problem for the community. We'd already tried asking him nicely to do a little more research for answers before posting, but that had no effect.

The strategy that finally worked is a perfect example of how to build a strong case on neutral, quantitative data. One of our developers, Brian Fitzpatrick, did some digging in the archives, and then sent the following message privately to a few developers. The offender (the third name on the list below, shown here as "J. Random") had very little history with the project, and had contributed no code or documentation. Yet he was the third most active poster on the mailing lists:

```
From: "Brian W. Fitzpatrick" <fitz@collab.net>
To: [... recipient list omitted for anonymity ...]
Subject: The Subversion Energy Sink
Date: Wed, 12 Nov 2003 23:37:47 -0600

In the last 25 days, the top 6 posters to the svn [dev|users] list have
been:

    294  kfogel@collab.net
    236  "C. Michael Pilato" <cmpilato@collab.net>
    220  "J. Random" <jrandom@problematic-poster.com>
    176  Branko #ibej <brane@xbc.nu>
    130  Philip Martin <philip@codematters.co.uk>
    126  Ben Collins-Sussman <sussman@collab.net>
```

```
I would say that five of these people are contributing to Subversion
hitting 1.0 in the near future.

I would also say that one of these people is consistently drawing time
and energy from the other 5, not to mention the list as a whole, thus
(albeit unintentionally) slowing the development of Subversion.  I did
not do a threaded analysis, but vgrepping my Subversion mail spool tells
me that every mail from this person is responded to at least once by at
least 2 of the other 5 people on the above list.

I think some sort of radical intervention is necessary here, even if we
do scare the aforementioned person away.  Niceties and kindness have
already proven to have no effect.

dev@subversion is a mailing list to facilitate development of a version
control system, not a group therapy session.

-Fitz, attempting to wade through three days of svn mail that he let
 pile up
```

Though it might not seem so at first, J. Random's behavior was a classic case of abusing project procedures. He wasn't doing something obvious like trying to filibuster a vote, but he was taking advantage of the mailing list's policy of relying on self-moderation by its members. We left it to each individual's judgement when to post and on what topics. Thus, we had no procedural recourse for dealing with someone who either did not have, or would not exercise, such judgement. There was no rule one could point to and say the fellow was violating it, yet everyone except him knew that his frequent posting was getting to be a serious problem.

Fitz's strategy was, in retrospect, masterful. He gathered damning quantitative evidence, but then distributed it discreetly, sending it first to a few people whose support would be key in any drastic action. They agreed that some sort of action was necessary, and in the end we called J. Random on the phone, described the problem to him directly, and asked him to simply stop posting. He never really did understand the reasons why; if he had been capable of understanding, he probably would have exercised appropriate judgement in the first place. But he agreed to stop posting, and the mailing lists became useable again. Part of the reason this strategy worked was, perhaps, the implicit threat that we could start restricting his posts via the moderation software normally used for preventing spam (see the section called “Spam Prevention” in Chapter 3, *Technical Infrastructure*). But the reason we were able to have that option in reserve was that Fitz had gathered the necessary support from key people first.

Handling Growth

The price of success is heavy in the open source world. As your software gets more popular, the number of people who show up looking for information increases dramatically, while the number of people able to provide information increases much more slowly. Furthermore, even if the ratio were evenly balanced, there is still a fundamental scalability problem with the way most open source projects handle communications. Consider mailing lists, for example. Most projects have a mailing list for general user questions—sometimes the list's name is "users", "discuss", "help", or something else. Whatever its name, the purpose of the list is always the same: to provide a place where people can get their questions answered, while others watch and (presumably) absorb knowledge from observing these exchanges.

These mailing lists work very well up to a few thousand users and/or a couple of hundred posts a day. But somewhere after that, the system starts to break down, because every subscriber sees every post; if the number of posts to the list begins to exceed what any individual reader can process in a day, the list

becomes a burden to its members. Imagine, for instance, if Microsoft had such a mailing list for Windows. Windows has hundreds of millions of users; if even one-tenth of one percent of them had questions in a given twenty-four hour period, then this hypothetical list would get hundreds of thousands of posts per day! Such a list could never exist, of course, because no one would stay subscribed to it. This problem is not limited to mailing lists; the same logic applies to IRC channels, other discussion forums, indeed to any system in which a group hears questions from individuals. The implications are ominous: the usual open source model of massively parallelized support simply does not scale to the levels needed for world domination.[3]

There will be no explosion when forums reach the breaking point. There is just a quiet negative feedback effect: people unsubscribe from the lists, or leave the IRC channel, or at any rate stop bothering to ask questions, because they can see they won't be heard in all the noise. As more and more people make this highly rational choice, the forum's activity will seem to stay at a manageable level. But it is staying manageable precisely because the rational (or at least, experienced) people have started looking elsewhere for information—while the inexperienced people stay behind and continue posting. In other words, one side effect of continuing to use unscalable communications models as a project grows is that the average *quality* of communications tends to go down. As the benefit/cost ratio of using high-population forums goes down, naturally those with the experience to do so start to look elsewhere for answers first. Adjusting communications mechanisms to cope with project growth therefore involves two related strategies:

1. Recognizing when particular parts of a forum are *not* suffering unbounded growth, even if the forum as a whole is, and separating those parts off into new, more specialized forums (i.e., don't let the good be dragged down by the bad).

2. Making sure there are many automated sources of information available, and that they are kept organized, up-to-date, and easy to find.

Strategy (1) is usually not too hard. Most projects start out with one main forum: a general discussion mailing list, on which feature ideas, design questions, and coding problems can all be hashed out. Everyone involved with the project is on the list. After a while, it usually becomes clear that the list has evolved into several distinct topic-based sublists. For example, some threads are clearly about development and design; others are user questions of the "How do I do X?" variety; maybe there's a third topic family centered around processing bug reports and enhancement requests; and so on. A given individual, of course, might participate in many different thread types, but the important thing is that there is not a lot of overlap between the types themselves. They could be divided into separate lists without causing harmful balkanization, because the threads rarely cross topic boundaries.

Actually doing this division is a two-step process. You create the new list (or IRC channel, or whatever it is to be), and then you spend whatever time is necessary gently nagging and reminding people to *use* the new forums appropriately. That latter step can last for weeks, but eventually people will get the idea. You simply have to make a point of always telling the sender when a post is sent to the wrong destination, and do so visibly, so that other people are encouraged to help out with routing. It's also useful to have a web page providing a guide to all the lists available; your responses can simply reference that web page and, as a bonus, the recipient may learn something about looking for guidelines before posting.

Strategy (2) is an ongoing process, lasting the lifetime of the project and involving many participants. Of course it is partly a matter of having up-to-date documentation (see the section called “Documentation” in Chapter 2, *Getting Started*) and making sure to point people there. But it is also much more than that; the sections that follow discuss this strategy in detail.

[3] An interesting experiment would be a probablistic mailing list, that sends each new thread-originating post to a random subset of subscribers, based on the approximate traffic level they signed up for, and keeps just that subset subscribed to the rest of the thread; such a forum could in theory scale without limit. If you try it, let me know how it works out.

Conspicuous Use of Archives

Typically, all communications in an open source project, except sometimes IRC conversations, are archived. The archives are public and searchable, and have referential stability: that is, once a given piece of information is recorded at a particular address (URL), it stays at that address forever.

Use those archives as much as possible, and as conspicuously as possible. Even when you know the answer to some question off the top of your head, if you think there's a reference in the archives that contains the answer, spend the time to dig it up and present it. Every time you do that in a publicly visible way, some people learn for the first time that the archives are there, and that searching in them can produce answers. Also, by referring to the archives instead of rewriting the advice, you reinforce the social norm against duplicating information. Why have the same answer in two different places? When the number of places it can be found is kept to a minimum, people who have found it before are more likely to remember what to search for to find it again. Well-placed references also contribute to the quality of search results in general, because they strengthen the targeted resource's ranking in Internet search engines.

There are times when duplicating information makes sense, however. For example, suppose there's a response already in the archives, not from you, saying:

```
It appears that your Scanley indexes have become frobnicated.  To
unfrobnicate them, run these steps:

1. Shut down the Scanley server.
2. Run the 'defrobnicate' program that ships with Scanley.
3. Start up the server.
```

Then, months later, you see another post indicating that someone's indexes have become frobnicated. You search the archives and come up with the old response above, but you realize it's missing some steps (perhaps by mistake, or perhaps because the software has changed since that post was written). The classiest way to handle this is to post a new, more complete set of instructions, and explicitly obsolete the old post by mentioning it:

```
It appears that your Scanley indexes have become frobnicated.  We
saw this problem back in July, and J. Random posted a solution at
http://blahblahblah/blah.  Below is a more complete description of
how to unfrobnicate your indexes, based on J. Random's instructions
but extending them a bit:

1. Shut down the Scanley server.
2. Become the user the Scanley server normally runs as.
3. As that user, run the 'defrobnicate' program on the indexes.
4. Run Scanley by hand to see if the indexes work now.
5. Restart the server.
```

(In an ideal world, it would be possible to attach a note to the old post, saying that there is newer information available and pointing to the new post. However, I don't know of any archiving software that offers an "obsoleted by" tag. This is another reason why creating dedicated web pages with answers to common questions is a good idea.)

Archives are probably most often searched for answers to technical questions, but their importance to the project goes well beyond that. If a project's formal guidelines are its statutory law, the archives are

its common law: a record of all decisions made and how they were arrived at. In any recurring discussion, it's pretty much obligatory nowadays to start with an archive search. This allows you to begin the discussion with a summary of the current state of things, anticipate objections, prepare rebuttals, and possibly discover angles you hadn't thought of. Also, the other participants will *expect* you to have done an archive search. Even if the previous discussions went nowhere, you should include pointers to them when you re-raise the topic, so people can see for themselves a) that they went nowhere, and b) that you did your homework, and therefore are probably saying something now that has not been said before.

Treat all resources like archives

All of the preceding advice applies to more than just mailing list archives. Having particular pieces of information at stable, conveniently findable addresses should be an organizing principle for all of the project's information. Let's take the project FAQ as a case study.

How do people use a FAQ?

1. They want to search in it for specific words and phrases.

2. They want to browse it, soaking up information without necessarily looking for answers to specific questions.

3. They expect search engines such as Google to know about the FAQ's content, so that searches can result in FAQ entries.

4. They want to be able to refer other people directly to specific items in the FAQ.

5. They want to be able to add new material to the FAQ, but note that this happens much less often than answers are looked up—FAQs are far more often read from than written to.

Point 1 implies that the FAQ should be available in some sort of textual format. Points 2 and 3 imply that the FAQ should be available as an HTML page, with point 2 additionally indicating that the HTML should be designed for readability (i.e., you'll want some control over its look and feel), and should have a table of contents. Point 4 means that each individual entry in the FAQ should be directly addresseable via a direct URL (e.g., using HTML IDs and named anchors, tags that allow people to reach a particular location on the page). Point 5 means the source files for the FAQ should be available in a convenient way (see the section called “Version everything” in Chapter 3, *Technical Infrastructure*), in a format that's easy to edit.

Formatting the FAQ like this is just one example of how to make a resource presentable. The same properties—direct searchability, availability to major Internet search engines, browsability, referential stability, and (where applicable) editability—apply to other web pages, to the source code tree, to the bug tracker, to Q&A forums, etc. It just happens that most mailing list archiving software long ago recognized the importance of these properties, which is why mailing lists tend to have this functionality natively, while other formats may require a little extra effort on the maintainer's part. Chapter 8, *Managing Participants* discusses how to spread that maintenance burden across many participants.

Codifying Tradition

As a project acquires history and complexity, the amount of data each new incoming participant must absorb increases. Those who have been with the project a long time were able to learn, and invent, the project's conventions as they went along. They will often not be consciously aware of what a huge body of tradition has accumulated, and may be surprised at how many missteps recent newcomers seem to make. Of course, the issue is not that the newcomers are of any lower quality than before; it's that they face a bigger acculturation burden than newcomers did in the past.

The traditions a project accumulates are as much about how to communicate and preserve information as they are about coding standards and other technical minutae. We've already looked at both sorts of standards, in the section called “Developer documentation” in Chapter 2, *Getting Started* and the section called “Writing It All Down” in Chapter 4, *Social and Political Infrastructure* respectively, and examples are given there. What this section is about is how to keep such guidelines up-to-date as the project evolves, especially guidelines about how communications are managed, because those are the ones that change the most as the project grows in size and complexity.

First, watch for patterns in how people get confused. If you see the same situations coming up over and over, especially with new participants, chances are there is a guideline that needs to be documented but isn't. Second, don't get tired of saying the same things over and over again, and don't *sound* like you're tired of saying them. You and other project veterans will have to repeat yourselves often; this is an inevitable side effect of the arrival of newcomers.

Every web page, every mailing list message, and every IRC channel should be considered advertising space—not for commercial advertisements, but for ads about your project's own resources. What you put in that space depends on the demographics of those likely to read it. An IRC channel for user questions, for example, is likely to get people who have never interacted with the project before—often someone who has just installed the software, and has a question he'd like answered immediately (after all, if it could wait, he'd have sent it to a mailing list instead, which would probably use less of his total time, although it would take longer for an answer to come back). Most people don't make a permanent investment in a support IRC channel; they'll show up, ask their question, and leave.

Therefore, the channel topic should be aimed at people looking for technical answers about the software *right now*, rather than at, say, people who might get involved with the project in a long term way and for whom community interaction guidelines might be more appropriate.

With mailing lists, the "ad space" is a tiny footer appended to every message. Most projects put subscription/unsubscription instructions there, and perhaps a pointer to the project's home page or FAQ page as well. You might think that anyone subscribed to the list would know where to find those things, and they probably do—but many more people than just subscribers see those mailing list messages. An archived post may be linked to from many places; indeed, some posts become so widely known that they eventually have more readers off the list than on it.

Formatting can make a big difference. For example, in the Subversion project, we were having limited success using the bug-filtering technique described in the section called “Pre-Filtering the Bug Tracker” in Chapter 3, *Technical Infrastructure*. Many bogus bug reports were still being filed by inexperienced people, and each time it happened, the filer had to be educated in exactly the same way as the 500 people before him. One day, after one of our developers had finally gotten to the end of his rope and flamed some poor user who didn't read the ticket tracker guidelines carefully enough, another developer decided this pattern had gone on long enough. He suggested that we reformat the ticket tracker front page so that the most important part, the injunction to discuss the bug on the mailing lists or IRC channels before filing, would stand out in huge, bold red letters, on a bright yellow background, centered prominently above everything else on the page. We did so (it's been reformatted a bit since then, but it's still very prominent—you can see the results at subversion.apache.org/reporting-issues.html [http://subversion.apache.org/reporting-issues.html]), and the result was a noticeable drop in the rate of bogus ticket filings. The project still gets them, of course—it always will—but the rate has slowed considerably, even as the number of users increases. The outcome is not only that the bug database contains less junk, but that those who respond to ticket filings stay in a better mood, and are more likely to remain friendly when responding to one of the now-rare bogus filings. This improves both the project's image and the mental health of its participants.

The lesson for us was that merely writing up the guidelines was not enough. We also had to put them where they'd be seen by those who need them most, and format them in such a way that their status as introductory material would be immediately clear to people unfamiliar with the project.

Static web pages are not the only venue for advertising the project's customs. A certain amount of interactive policing (in the friendly-reminder sense, not the handcuffs-and-jail sense) is also required. All peer review, even the commit reviews described in the section called “Practice Conspicuous Code Review” in Chapter 2, *Getting Started*, should include review of people's conformance or non-conformance with project norms, especially with regard to communications conventions.

Another example from the Subversion project: we settled on a convention of "r12908" to mean "revision 12908 in the version control repository." The lower-case "r" prefix is easy to type, and because it's half the height of the digits, it makes an easily-recognizable block of text when combined with the digits. Of course, settling on the convention doesn't mean that everyone will begin using it consistently right away. Thus, when a commit mail comes in with a log message like this:

```
------------------------------------------------------------------------
r12908 | qsimon | 2005-02-02 14:15:06 -0600 (Wed, 02 Feb 2005) | 4 lines

Patch from J. Random Contributor <jrcontrib@gmail.com>

* trunk/contrib/client-side/psvn/psvn.el:
  Fixed some typos from revision 12828.
------------------------------------------------------------------------
```

...part of reviewing that commit is to say "By the way, please use 'r12828', not 'revision 12828' when referring to past changes." This isn't just pedantry; it's important as much for automatic parsability as for human readership.

By following the general principle that there should be canonical referral methods for common entities, and that these referral methods should be used consistently everywhere, the project in effect exports certain standards. Those standards enable people to write tools that present the project's communications in more useable ways—for example, a revision formatted as "r12828" could be transformed into a live link into the repository browsing system. This would be harder to do if the revision were written as "revision 12828", both because that form could be divided across a line break, and because it's less distinct (the word "revision" will often appear alone, and groups of numbers will often appear alone, whereas the combination "r12828" can only mean a revision number). Similar concerns apply to ticket numbers, FAQ items, etc.[4]

(Note that for Git commit IDs, the widely-accepted standard syntax is "`commit c03dd89305`, that is, the word "commit", followed by a space, followed by the first 8-10 characters of the commit hash. Some very busy projects have standardized on 12 characters, to avoid collisions; the only time all 40 characters of the hash are used is in non-human-readable contexts, like saving a commit ID in an automated release-tracking system or something.)

Even for entities where there is not an obvious short, canonical form, people should still be encouraged to provide key pieces of information consistently. For example, when referring to a mailing list message, don't just give the sender and subject; also give the archive URL *and* the Message-ID header. The last allows people who have their own copy of the mailing list (people sometimes keep offline copies, for example to use on a laptop while traveling) to unambiguously identify the right message in a search even if they don't have access to the archives. The sender and subject wouldn't be enough, because the same person might make several posts in the same thread, even on the same day.

The more a project grows, the more important this sort of consistency becomes. Consistency means that everywhere people look, they see the same patterns being followed, and start to follow those patterns

[4] A more extended example of the kinds of benefits such standards make possible is the Contribulyzer example mentioned in the section called “The Automation Ratio”, in Chapter 8, *Managing Participants*.

themselves. This, in turn, reduces the number of questions they need to ask. The burden of having a million readers is no greater than that of having one; scalability problems start to arise only when a certain percentage of those readers ask questions. As a project grows, therefore, it must reduce that percentage by increasing the density and findability of information, so that any given person is more likely to find what he needs without having to ask.

Choose the Right Forum

One of the trickiest things about managing an open source project is getting people to be thoughtful about which forum they choose for different kinds of communications. It's tricky partly because it's not immediately obvious that it matters. During any given conversation, the participants are mostly concerned with what the people involved are saying, and won't usually stop to think about whether or not the forum itself gives others who *might* want to take part the opportunity to do so.

For example, a real-time forum like IRC is terrific for quick questions, for opportunistic synchronization of work, for reminding someone of something they promised to do, etc. But it's not a good forum for making decisions that affect the whole project, because the people who take part in a conversation in IRC are just whoever happened to be in the channel at that moment — it's very dependent on work schedules, time zones, etc. On the other hand, the development mailing list is a great place for making formal project-wide decisions, since every interested party will have an opportunity to see and respond to the relevant posts, even though it's not as well-suited to quick, real-time interactions as IRC is.

Another example comes up frequently in bug tracker usage, especially in the last few years as bug trackers have become so well integrated with email. Sometimes people will be drawn into a discussion in a bug ticket[5] and because they simply see project-related emails coming in to their email client, they treat the discussion as though it's happening on the real development list. But it's not: anyone who wasn't watching that bug and wasn't explicitly invited into the conversation won't even be aware it's happening. If things are discussed in that bug ticket that go beyond the scope of just that one bug, they'll be discussed without input from people who should at least have had a chance to participate.

The solution to this is to encourage conscious, intentional forum changes. If a discussion starts to get into questions beyond the scope of its original forum, then at some point someone involved should ask that the conversation move over to the main development list or some other wider forum.

It's not enough for you to do this on your own. You have to create a culture where it's normal for everyone to do it, so everyone thinks about forum appropriateness as a matter of course, and feels comfortable raising questions of forum whenever necessary in any discussion. Obviously, documenting the practice will help (see the section called “Writing It All Down” in Chapter 4, *Social and Political Infrastructure*), but you'll probably also need to remind people of it often, especially when your project is starting out. A good rule of thumb is: if the conversation looks convergent, then it's okay to keep it in the bug ticket or other original forum. But if it looks likely to diverge for a while (e.g., widening into philosophical issues about how the software should behave, or raising design issues that go beyond just the one bug) before it converges, then take the discussion to a broader forum, usually the development mailing list.

Cross-Link Between Forums

When a discussion moves from one place to another, cross-link between the old and new place. For example, if discussion moves from the ticket tracker to the mailing list, link to the mailing list thread from the ticket, and mention the original ticket at the start of the new list thread. It's important for someone following the ticket to be able to reach the later discussion; it's also important for someone who en-

[5] For example, on GitHub, simply mentioning someone's GitHub account name with an @-sign (e.g., `@kfogel`) in a comment on a ticket will cause that person to be added to the email thread associated with that ticket.

counters the ticket a year later to be able to follow to where the conversation went to in the mailing list archives. The person who does the move may find this cross-linking slightly laborious, but open source is fundamentally a writer-responsible culture. It's more important to make things easy for the tens or hundreds of people who may read the bug than for the three or five people writing about it.

It's also fine to take important conclusions or summaries from the list discussion and paste them into the ticket at the end, if that will make things convenient for readers. A common idiom is to move discussion to the mailing list, put a link to that thread in the ticket, and then when the discussion finishes, paste the final summary into the ticket (along with a link to the message containing that summary), so someone browsing the ticket later can easily see what conclusion was reached without having to click to somewhere else or do detective work. Note that the usual "two masters" data duplication problem does not exist here, because both archives and ticket comments are usually static, unchangeable data anyway.

Publicity

In free software, there is a fairly smooth continuum between purely internal discussions and public relations statements. This is partly because the target audience is always ill-defined: given that most or all posts are publicly accessible, the project doesn't have full control over the impression the world gets. Someone—say, a Hacker News [https://news.ycombinator.com/] poster or slashdot.org [https://slashdot.org/] editor—may draw millions of readers' attention to a post that no one ever expected to be seen outside the project. This is a fact of life that all open source projects live with, but in practice, the risk is usually small. In general, the announcements that the project most wants publicized are the ones that will be most publicized, assuming you use the right mechanisms to indicate relative newsworthiness to the outside world.

Announcing Releases and Other Major Events

For major announcements, there tend to be a few main channels of distribution, on which announcements should be made as nearly simultaneously as possible:

1. Your project's front page is probably seen by more people than any other part of the project. If you have a really major announcement, put a blurb there. The blurb should be a very brief synopsis that links to the press release (see below) for more information.

2. At the same time, you should also have a "News" or "Press Releases" area of the web site, where the announcement can be written up in detail. Part of the purpose of a press release is to provide a single, canonical "announcement object" that other sites can link to, so make sure it is structured accordingly: either as one web page per release, as a discrete blog entry, or as some other kind of entity that can be linked to while still being kept distinct from other press releases in the same area.

3. Make sure the announcement gets broadcast by any relevant Twitter handles, and goes out on any news channels or RSS feeds. (The latter may happen automatically when you publish announcement, depending on how things are set up at your site.)

4. Post to forums as appropriate, in the manner described in the section called “Announcing”) in Chapter 2, *Getting Started*.

5. Send a mail to your project's announcement mailing list. This list's name should actually be "announce", that is, `announce@yourprojectdomain.org`, because that's a fairly standard convention now, and the list's charter should make it clear that it is very low-traffic, reserved for major project announcements. Most of those announcements will be about new releases of the software, but occasionally other events, such as a fundraising drive, the discovery of a security vulnerability (see the section called “Announcing Security Vulnerabilities”) later in this chapter, or a major shift

in project direction may be posted there as well. Because it is low traffic and used only for important things, the announce list typically has the highest subscribership of any mailing list in the project (of course, this means you shouldn't abuse it—consider carefully before posting). To avoid random people making announcements, or worse, spam getting through, the announce list must always be moderated.

Try to make the announcements in all these places at the same time, as nearly as possible. People might get confused if they see an announcement on the mailing list but then don't see it reflected on the project's home page or in its press releases area. If you get the various changes (emails, web page edits, etc.) queued up and then send them all in a row, you can keep the window of inconsistency very small.

For a less important event, you can eliminate some or all of the above outlets. The event will still be noticed by the outside world in direct proportion to its importance. For example, while a new release of the software is a major event, merely setting the date of the next release, while still somewhat newsworthy, is not nearly as important as the release itself. Setting a date is worth an email to the daily mailing lists (not the announce list), and an update of the project's timeline or status web page, but no more.

However, you might still see that date appearing in discussions elsewhere on the Internet, wherever there are people interested in the project. People who are lurkers on your mailing lists, just listening and never saying anything, are not necessarily silent elsewhere. Word of mouth gives very broad distribution; you should count on it, and construct even minor announcements in such a way as to encourage accurate informal transmission. Specifically, posts that you expect to be quoted should have a clearly meant-to-be-quoted portion, just as though you were writing a formal press release. For example:

> *Just a progress update: we're planning to release version 2.0 of Scanley in mid-August 2005. You can always check http://www.scanley.org/status.html for updates. The major new feature will be regular-expression searches.*
>
> *Other new features include: ... There will also be various bugfixes, including: ...*

The first paragraph is short, gives the two most important pieces of information (the release date and the major new feature), and a URL to visit for further news. If that paragraph is the only thing that crosses someone's screen, you're still doing pretty well. The rest of the mail could be lost without affecting the gist of the content. Of course, sometimes people will link to the entire mail anyway, but just as often, they'll quote only a small part. Given that the latter is a possibility, you might as well make it easy for them, and in the bargain get some influence over what gets quoted.

Announcing Security Vulnerabilities

Handling a security vulnerability is different from handling any other kind of bug report. In free software, doing things openly and transparently is normally almost a religious credo. Every step of the standard bug-handling process is visible to all who care to watch: the arrival of the initial report, the ensuing discussion, and the eventual fix.

Security bugs are different. They can compromise users' data, and possibly users' entire computers. To discuss such a problem openly would be to advertise its existence to the entire world—including to all the parties who might make malicious use of the bug. Even merely committing a fix effectively announces the bug's existence (there are potential attackers who watch the commit logs of public projects, systematically looking for changes that indicate security problems in the pre-change code). Most open source projects have settled on approximately the same set of steps to handle this conflict between openness and secrecy, based on the these basic guidelines:

1. Don't talk about the bug publicly until a fix is available; then supply the fix (packaged as a release) at exactly the same moment you announce the bug.

2. Come up with that fix as fast as you can—especially if someone outside the project reported the bug, because then you know there's at least one person outside the project who is able to exploit the vulnerability.

In practice, those principles lead to a fairly standardized series of steps, which are described in the sections below.

Receive the report

Obviously, a project needs the ability to receive security bug reports from anyone. But the regular bug reporting channels won't do, because they can be watched by anyone too. Therefore, have a separate mailing list or contact form for receiving security bug reports. That forum must not have publicly readable archives, and its subscribership must be strictly controlled—only long-time, trusted developers can be on the list, and people whom such developers have consensus that they trust[6]. If you need a formal definition of "trusted developer", you can use "anyone who has had commit access for two years or more" or something like that, to avoid favoritism. This is the group that will handle security bugs.

Ideally, that reporting gateway should not be spam-protected or moderated, since you don't want an important report to get filtered out or delayed just because no moderators happened to be online that weekend. If you do use automated spam-protection software, try to configure it with high-tolerance settings; it's better to let a few spams through than to miss a vulnerability report.

The submission mechanism should itself be secure. That is, if it is a contact form, it should be on an `https://` (TLS-protected) page, or if it is an email address, there should be a well-advertised public key (digitally signed by as many of the core developers as possible) so people can send encrypted mails to that address.[7] A web form submission or an email sent to your project may travel over many Internet hops on its way there; you have no reason to trust whoever runs those intermediate servers, and there is a flourishing market for new security vulnerabilities. Assume the worst and design accordingly.

Develop the fix quietly

So what does the security list do when it receives a report? The first task is to evaluate the problem's severity and urgency:

1. How serious is the vulnerability? Does it allow a malicious attacker to take over the computer of someone who uses your software? Or does it, say, merely leak information about the sizes of some of their files?

2. How easy is it to exploit the vulnerability? Can an attack be scripted, or does it require circumstantial knowledge, educated guessing, and luck?

3. *Who* reported the problem to you? The answer to this question doesn't change the nature of the vulnerability, of course, but it does give you an idea of how many other people might know about it. If the report comes from one of the project's own developers, you can breathe a little easier (but only a little), because you can trust them not to have told anyone else about it. On the other hand, if it came in an email from `anonymous14@globalhackerz.net`, then you'd better act as fast as you can. The person did you a favor by informing you of the problem at all, but you have no idea how many other people she's told, or how long she'll wait before exploiting the vulnerability on live installations.

[6] E.g., a release manager who maybe isn't a core developer but who is already trusted to roll releases anyway. I've seen cases where companies who had been long involved in a project had managers as members of its security group, even though those managers had never committed a line of code, because by common consent the project's maintainers trusted them and felt it was to the project's benefit for them to see vulnerability reports as soon as possible. There is no one rule that will be appropriate for all projects, but in general, the core maintainers should follow the principle that anyone who receives security reports must be trustable both in terms of intention and in terms of their technical ability to not accidentally leak information (e.g., someone whose email gets hacked regularly should probably not be on the security list).

[7] If you don't know what all of these terms mean, find people you trust who do and get them to help your project. Handling security vulnerablities competently requiries a working knowledge of these concepts.

Note that the difference we're talking about here is often just a narrow range between *urgent* and *extremely urgent*. Even when the report comes from a known, friendly source, there could be other people on the Net who discovered the bug long ago and just haven't reported it. The only time things aren't urgent is when the bug inherently does not compromise security very severely.

The "`anonymous14@globalhackerz.net`" example is not facetious, by the way. You really may get bug reports from identity-cloaked people who, by their words and behavior, never quite clarify whether they're on your side or not. It doesn't matter: if they've reported the security hole to you, they'll feel they've done you a good turn, and you should respond in kind. Thank them for the report, give them a date on or before which you plan to release a public fix, and keep them in the loop. Sometimes they may give *you* a date—that is, an implicit threat to publicize the bug on a certain date, whether you're ready or not. This may feel like a bullying power play, but it's more likely a preëmptive action resulting from past disappointment with unresponsive software producers who didn't take security reports seriously enough. Either way, you can't afford to tick this person off. After all, if the bug is severe, she has knowledge that could cause your users big problems. Treat such reporters well, and hope that they treat you well.

Another frequent reporter of security bugs is the security professional, someone who audits code for a living and keeps up on the latest news of software vulnerabilities. These people usually have experience on both sides of the fence—they've both received and sent reports, probably more than most developers in your project have. They too will usually give a deadline for fixing a vulnerability before going public. The deadline may be somewhat negotiable, but that's up to the reporter; deadlines have become recognized among security professionals as pretty much the only reliable way to get organizations to address security problems promptly. So don't treat the deadline as rude; it's a time-honored tradition, and there are good reasons for it. Negotiate if you absolutely must, but remember that the reporter holds all the cards.

Once you know the severity and urgency, you can start working on a fix. There is sometimes a tradeoff between doing a fix elegantly and doing it speedily; this is why you must agree on the urgency before you start. Keep discussion of the fix restricted to the security list members, of course, plus the original reporter (if she wants to be involved) and any developers who need to be brought in for technical reasons.

Do not commit the fix to any public repository before the go-public date. If you were to commit it publicly, even with an innocent-looking log message, someone might notice and understand the change. You never know who is watching your repository and why they might be interested. Turning off commit emails wouldn't help; first of all, the gap in the commit mail sequence would itself look suspicious, and anyway, the data would still be in the repository. Just do all development in some private place known only to the people already aware of the bug.

CVE numbers

You may have seen a *CVE number* associated with a particular security problems — e.g., a number like "CVE-2014-0160", where the first numeric part is the year, and the second is an increasing sequence number (it may exceed four digits if more than 10,000 numbers are handed out in a given year).

A CVE number is an entry in the "Common Vulnerabilities and Exposures" list maintained at cve.mitre.org [https://cve.mitre.org/]. The purpose of the list is to a provide standardized name for every known computer security problem, so that everyone has a unique, canonical name to use when discussing it, and a central place to go to find out more information.[8]

[8] In the past, a CVE number would start out as a CAN number ("CAN" for "candidate") until it was approved for inclusion in the official list, at which point the "CAN" would be replaced with "CVE" while the number portion remained the same. However, nowadays they are just assigned a "CVE-" prefix from the start, although that prefix does not guarantee that the vulnerability will be included in the official list. (For example, it might be later discovered to be a duplicate of an existing CVE, in which case the earlier one — the lower number — should be used.)

A CVE entry does not itself contain a full description of the bug and how to protect against it. Instead, it contains a brief summary, and a list of references to external resources (such as a announcement post from the project in question) where people can go to get more detailed information. The real purpose of cve.mitre.org [https://cve.mitre.org/] is to provide a well-organized space in which every vulnerability has a single name, and people have a clear route to get more data about it. See cve.mitre.org/cgi-bin/cvename.cgi?name=2014-0160 [https://cve.mitre.org/cgi-bin/cvename.cgi?name=2014-0160] for an example of an entry. Note that the references can be very terse, with sources appearing as cryptic abbreviations. A key to those abbreviations is at cve.mitre.org/data/refs/refkey.html [https://cve.mitre.org/data/refs/refkey.html].

If your vulnerability meets the criteria, you may wish to obtain a CVE number for it. You can request one using the instructions at cve.mitre.org/cve/request_id.html [https://cve.mitre.org/cve/request_id.html], but if there is someone in your project who has already obtained CVE numbers, or who knows someone who has, let them do it. The CVE Editorial Board gets a lot of submissions, many of them spurious or poorly written submissions; by approaching them through a trusted source, you are saving them time and possibly getting your CVE number assigned more quickly. The other advantage of doing it this way is that somewhere along the chain, someone may know enough to tell you that a) it wouldn't count as a vulnerability or exposure according to MITRE's criteria, so there is no point submitting it, or b) the vulnerability already *has* a CVE number. The latter can happen if the bug has already been published on another security advisory list, for example at www.cert.org [https://www.cert.org/] or on the BugTraq mailing list at www.securityfocus.com [http://www.securityfocus.com/]. (If that happened without your project hearing about it, then you should worry what else might be going on that you don't know about.)

If you get a CVE number at all, you usually want to get it in the early stages of your bug investigation, so that all further communications can refer to that number. CVE entries are embargoed until the go-public date: the number will be reserved, but MITRE will allow some time (within reason) before revealing information about the vulnerability — so make sure you or your intermediary communicate clearly with the CVE Editorial Board about how long you need before publicly announcing.

See cve.mitre.org [https://cve.mitre.org/] for more information about the CVE process. See also debian.org/security/cve-compatibility [https://www.debian.org/security/cve-compatibility] for a particularly clear exposition of one open source project's use of CVE numbers, and see securityblog.redhat.com/2013/01/30/a-minimal-security-response-process [https://securityblog.redhat.com/2013/01/30/a-minimal-security-response-process/] for a good writeup of a minimal security response process, from a security engineer at RedHat.

Pre-notification

Once your security response team (that is, those developers who are on the security mailing list, or who have been brought in to deal with a particular report) has a fix ready, you need to decide how to distribute it.

If you simply commit the fix to your repository, or otherwise announce it to the world, you effectively force everyone using your software to upgrade immediately or risk being hacked. It is sometimes appropriate, therefore, to do *pre-notification* for certain important users. This is particularly true with client/server software, where there may be well-known services using your software and who are tempting targets for attackers. Those service's administrators would appreciate having an extra day or two to do the upgrade, so that they are already protected by the time the exploit becomes public knowledge.

Pre-notification simply means sending mails to those administrators before the go-public date, telling them of the vulnerability and how to fix it. You should send pre-notification only to people you trust to be discreet with the information, and with whom you can communicate securely. That is, the qualification for receiving pre-notification is threefold: the recipient must run a large, important service where a compromise would be a serious matter; the recipient must be known to be someone who won't blab about the security problem before the go-public date; and you must have a way (such as via GnuPG

[https://gnupg.org/]-encrypted email) to communicate securely with the recipient, so that any eavesdroppers between you and your recipient can't read the message.[9]

Pre-notification should be done via secure means. If email, then encrypt it, for the same reasons explained in the section called “Receive the report”, but if you have a phone number or other out-of-band secure way to contact the administrator, use that. When sending encrypted pre-notification emails, send them individually (one at a time) to each recipient. Do *not* send to the entire list of recipients at once, because then they would see each others' names—meaning that you would essentially be alerting each recipient to the fact that each *other* recipient may have a security hole in her service. Sending it to them all via blind CC (BCC) isn't a good solution either, because some admins protect their inboxes with spam filters that either block or reduce the priority of BCC'd mail, since so much spam is sent via BCC.

Here's a sample pre-notification mail:

```
From: Your Name Here
To: admin@large-famous-server.com
Reply-to: Your Name Here (not the security list's address)
Subject: Confidential important notification.

[[[ BEGIN ENCRYPTED MAIL ]]]

This email is a confidential pre-notification of a security alert
in the Scanley server software.

Please *do not forward* any part of this mail to anyone.  The public
announcement is not until May 19th, and we'd like to keep the
information embargoed until then.

You are receiving this mail because (we think) you run a Scanley
server, and would want to have it patched before this security hole is
made public on May 19th.

References:
===========

   CVE-2015-892346: Scanley stack overflow in queries

Vulnerability:
==============

   The server can be made to run arbitrary commands if the server's
   locale is misconfigured and the client sends a malformed query.

Severity:
=========

   Very severe; can involve arbitrary code execution on the server.

Workarounds:
============

   Setting the 'natural-language-processing' option to 'off' in
```

[9]Remember that Subject lines in emails aren't encrypted, so don't put too much information about the vulnerability in the Subject line.

```
   scanley.conf closes this vulnerability.

Patch:
======

   The patch below applies to Scanley 3.0, 3.1, and 3.2.

   A new public release (Scanley 3.2.1) will be made on or just before
   May 19th, so that it is available at the same time as this
   vulnerability is made public.  You can patch now, or just wait for
   the public release.  The only difference between 3.2 and 3.2.1 will
   be this patch.

[...patch goes here...]
```

If you have a CVE number, include it in the pre-notification (as shown above), even though the information is still embargoed and therefore the corresponding MITRE page will show nothing at the time of pre-notification. Including the CVE number allows the recipient to know with certainty that the bug they were pre-notified about is the same one they later hear about through public channels, so they don't have to worry whether further action is necessary or not, which is precisely the point of CVE numbers.

Distribute the fix publicly

The last step in handling a security bug is to distribute the fix publicly. In a single, comprehensive announcement, you should describe the problem, give the CVE number if any, describe how to work around it, and how to permanently fix it. Usually "fix" means upgrading to a new version of the software, though sometimes it can mean applying a patch, particularly if the software is normally run in source form anyway. If you do make a new release, it should differ from some existing release by exactly the security patch. That way, conservative admins can upgrade without worrying about what else they might be affecting; they also don't have to worry about future upgrades, because the security fix will be in all future releases as a matter of course. (Details of release procedures are discussed in the section called “Security Releases” in Chapter 7, *Packaging, Releasing, and Daily Development*.)

Whether or not the public fix involves a new release, do the announcement with roughly the same priority as you would a new release: send a mail to the project's `announce` list, make a new press release, etc. While you should never try to play down the existence of a security bug out of concern for the project's reputation, you may certainly set the tone and prominence of a security announcement to match the actual severity of the problem. If the security hole is just a minor information exposure, not an exploit that allows the user's entire computer to be taken over, then it may not warrant a lot of fuss. You may even decide not to distract the `announce` list with it. After all, if the project cries wolf every time, users might end up thinking the software is less secure than it actually is, and also might not believe you when you have a really big problem to announce. See cve.mitre.org/about/terminology.html [https://cve.mitre.org/about/terminology.html] for a good introduction to the problem of judging and describing severity.

In general, if you're unsure how to treat a security problem, find someone with experience and talk to them about it. Assessing and handling vulnerabilities is very much an acquired skill, and it's easy to make missteps the first few times.

See also the Apache Software Foundation guidelines on handling security vulnerabilities at apache.org/security/committers.html [https://www.apache.org/security/committers.html]. They are an excellent checklist you can compare against to see if you're doing everything carefully.

Chapter 7. Packaging, Releasing, and Daily Development

This chapter is about how free software projects package and release their software, and how overall development patterns organize around those goals.

A major difference between open source projects and proprietary ones is the lack of centralized control over the development team. When a new release is being prepared, this difference is especially stark: if a single corporation manages the entire development team, it can ask them to focus on an upcoming release, putting aside new feature development and non-critical bug fixing until the release is done. But open source developer communities are are not so monolithic. People work on the project for all sorts of reasons, and those not interested in helping with a given release still want to continue regular development work while the release is going on. Because development doesn't stop, open source release processes tend to take longer, but be less disruptive, than commercial release processes. It's a bit like highway repair. There are two ways to fix a road: you can shut it down completely, so that a repair crew can swarm all over it at full capacity until the problem is solved, or you can work on a couple of lanes at a time, while leaving the others open to traffic. The first way is very efficient *for the repair crew*, but not for anyone else—the road is entirely shut down until the job is done. The second way involves much more time and trouble for the repair crew (now they have to work with fewer people and less equipment, in cramped conditions, with flaggers to slow and direct traffic, etc.), but at least the road remains useable, albeit not at full capacity.

Open source projects tend to work the second way. In fact, for a mature piece of software with several different release lines being maintained simultaneously, the project is sort of in a permanent state of minor road repair. There are always a couple of lanes closed; a consistent but low level of background inconvenience is always being tolerated by the development group as a whole, so that releases get made on a regular schedule.

The model that makes this possible generalizes to more than just releases. It's the principle of parallelizing tasks that are not mutually interdependent—a principle that is by no means unique to open source development, of course, but one which open source projects implement in their own particular way. They cannot afford to annoy either the roadwork crew or the regular traffic too much, but they also cannot afford to have people dedicated to standing by the orange cones and flagging traffic along. Thus they gravitate toward processes that have flat, constant levels of administrative overhead, rather than peaks and valleys. Developers are generally willing to work with small but consistent amounts of inconvenience; the predictability allows them to come and go without worrying about whether their schedule will clash with what's happening in the project. But if the project were subject to a master schedule in which some activities excluded other activities, the result would be a lot of developers sitting idle a lot of the time—which would be not only inefficient but boring, and therefore dangerous, in that a bored developer is likely to soon be an ex-developer.

Release work is usually the most noticeable non-development task that happens in parallel with development, so the methods described in the following sections are geared mostly toward enabling releases. However, note that they also apply to other parallelizable tasks, such as translations and internationalization, broad API changes made gradually across the entire codebase, etc.

Release Numbering

Before we talk about how to make a release, let's look at how to name releases, which requires knowing what releases actually mean to users. A release means that:

- Some old bugs have been fixed. This is probably the one thing users can count on being true of every release.

- New bugs have been added. This too can usually be counted on, except sometimes in the case of security releases or other one-offs (see the section called “Security Releases” later in this chapter).

- New features may have been added.

- New configuration options may have been added, or the meanings of old options may have changed subtly. The installation procedures may have changed slightly since the last release too, though one always hopes not.

- Incompatible changes may have been introduced, for example such that the data formats used by older versions of the software are no longer useable without undergoing some sort of (possibly manual) one-way conversion step.

As you can see, not all of these are good things. This is why experienced users approach new releases with some trepidation, especially when the software is mature and was already mostly doing what they wanted (or thought they wanted). Even the arrival of new features is a mixed blessing, in that it may mean the software will now behave in unexpected ways.

The purpose of release numbering, therefore, is twofold: obviously the numbers should unambiguously communicate the ordering of releases (i.e., by looking at any two releases' numbers, one can know which came later), but also they should indicate as compactly as possible the degree and nature of the changes in the release.

Some projects just need release identifiers, not release numbers.

The advice in the rest of this section only applies to projects where release number semantics matter. Use your judgement: if your project isn't offering API predictability anyway, or if it practices continuous development with auto-deployment in the way that (for example) some Javascript projects do, then maybe you can get away with just letting git commit IDs double as release identifiers, or with some other similarly lightweight method. Just make sure to consider the question carefully, and to base your decision on how users actually deploy and upgrade the software. When it comes to release numbering, it's better to be overly strict than overly lax. Remember that the project's core developers are not the main audience for release numbers; those developers already know what's happening in the project, what APIs have changed, etc. Release numbers are most important for people who *don't* follow the project on a daily basis, and who are therefore naturally underrepresented in project discussions about how strictly to adhere to a release numbering scheme. If you believe in the users, stand up for them!

All that in a number? Well, more or less, yes. Release numbering strategies are one of the oldest bikeshed discussions around (see the section called “The Smaller the Topic, the Longer the Debate” in Chapter 6, *Communications*), and the world is unlikely to settle on a single, complete standard anytime soon. However, a few good strategies have emerged, along with one universally agreed-on principle: *be consistent*. Pick a numbering scheme, document it, and stick with it. Your users will thank you.

Release Number Components

This section describes the formal conventions of release numbering in detail, and assumes very little prior knowledge. It is intended mainly as a reference. If you're already familiar with these conventions, you can skip this section.[1]

Release numbers are groups of digits separated by dots:

Scanley 2.3
Singer 5.11.4

...and so on. The dots are *not* decimal points, they are merely separators; "5.3.9" would be followed by "5.3.10". A few projects have occasionally hinted otherwise, most famously the Linux kernel with its "0.95", "0.96"... "0.99" sequence leading up to Linux 1.0, but the convention that the dots are not decimals is now firmly established and should be considered a standard. There is no limit to the number of components (digit portions containing no dots), but most projects do not go beyond three or four. The reasons why will become clear later.

In addition to the numeric components, projects sometimes tack on a descriptive label such as "Alpha" or "Beta" (see Alpha and Beta), for example:

Scanley 2.3.0 (Alpha)
Singer 5.11.4 (Beta)

An Alpha or Beta qualifier means that this release *precedes* a future release that will have the same number without the qualifier. Thus, "2.3.0 (Alpha)" leads eventually to "2.3.0". In order to allow several such candidate releases in a row, the qualifiers themselves can have meta-qualifiers. For example, here is a series of releases in the order that they would be made available to the public:

Scanley 2.3.0 (Alpha 1)
Scanley 2.3.0 (Alpha 2)
Scanley 2.3.0 (Beta 1)
Scanley 2.3.0 (Beta 2)
Scanley 2.3.0 (Beta 3)
Scanley 2.3.0

Notice that when it has the "Alpha" qualifier, Scanley "2.3" is written as "2.3.0". The two numbers are equivalent—trailing all-zero components can always be dropped for brevity—but when a qualifier is present, brevity is out the window anyway, so one might as well go for completeness instead.

Other qualifiers in semi-regular use include "Stable", "Unstable", "Development", and "RC" (for "Release Candidate"). The most widely used ones are still "Alpha" and "Beta", with "RC" running a close third place, but note that "RC" always includes a numeric meta-qualifier. That is, you don't release "Scanley 2.3.0 (RC)", you release "Scanley 2.3.0 (RC 1)", followed by RC2, etc.

Those three labels, "Alpha", "Beta", and "RC", are pretty widely known now, and I don't recommend using any of the others, even though the others might at first glance seem like better choices because they are normal words, not jargon. But people who install software from releases are already familiar with the big three, and there's no reason to do things gratuitously differently from the way everyone else does them.

[1]Since the first edition of this book was published, these conventions have been formalized, in a more detailed way, as the *Semantic Versioning* standard and are maintained at semver.org [http://semver.org/]. As of this writing in July 2015 the latest version of the standard is 2.0.0. The recommendations made there seem to be roughly the same as the first strategy documented here, except that Semantic Versioning apparently does not include the forward-compatibility requirement for increments in the micro (patch) number.

Although the dots in release numbers are not decimal points, they do indicate place-value significance. All "0.X.Y" releases precede "1.0" (which is equivalent to "1.0.0", of course). "3.14.158" immediately precedes "3.14.159", and non-immediately precedes "3.14.160" as well as "3.15.anything", and so.

A consistent release numbering policy enables a user to look at two release numbers for the same piece of software and tell, just from the numbers, the important differences between those two releases. In a typical three-component system, the first component is the *major number*, the second is the *minor number*, and the third is the *micro number* (sometimes also called the "patch" number). For example, release "2.10.17" is the eighteenth micro release (or patch release) in the eleventh minor release line within the second major release series[2]. The words "line" and "series" are used informally here, but they mean what one would expect: a major series is simply all the releases that share the same major number, and a minor series (or minor line) consists of all the releases that share the same minor *and* major number. That is, "2.4.0" and "3.4.1" are not in the same minor series, even though they both have "4" for their minor number; on the other hand, "2.4.0" and "2.4.2" are in the same minor line, though they are not adjacent if "2.4.1" was released between them.

The meanings of these numbers themselves are also roughly what you'd expect: an increment of the major number indicates that major changes happened; an increment of the minor number indicates minor changes; and an increment of the micro number indicates really trivial changes. Some projects add a fourth component, usually called the *patch number*, for especially fine-grained control over the differences between their releases (confusingly, other projects use "patch" as a synonym for "micro" in a three-component system). There are also projects that use the last component as a *build number*, incremented every time the software is built and representing no change other than that build. This helps the project link every bug report with a specific build, and is probably most useful when binary packages are the default method of distribution.

Although there are many different conventions for how many components to use, and what the components mean, the differences tend to be minor—you get a little leeway, but not a lot. The next two sections discuss some of the most widely used conventions.

The Simple Strategy

Most projects have rules about what kinds of changes are allowed into a release if one is only incrementing the micro number, different rules for the minor number, and still different ones for the major number. There is no set standard for these rules yet, but here I will describe a policy that has been used successfully by multiple projects. You may want to just adopt this policy in your own project, but even if you don't, it's still a good example of the kind of information release numbers should convey. This policy is adapted from the numbering system used by the APR project, see apr.apache.org/versioning.html [https://apr.apache.org/versioning.html].

1. Changes to the micro number only (that is, changes within the same minor line) must be both forward- and backward-compatible. The changes should be bug fixes only, or very small enhancements to existing features. New features should not be introduced in a micro release.

2. Changes to the minor number (that is, within the same major line) must be backward-compatible, but not necessarily forward-compatible. It's normal to introduce new features in a minor release, but usually not too many new features at once.

3. Changes to the major number mark compatibility boundaries. A new major release can be forward- and backward-incompatible. A major release is expected to have new features, and may even have entire new feature sets.

What *backward-compatible* and *forward-compatible* mean, exactly, depends on what your software does, but in context they are usually not open to much interpretation. For example, if your project is a

[2]Not seventeenth and tenth, because numbering starts from 0, not 1.

client/server application, then "backward-compatible" means that upgrading the server to 2.6.0 should not cause any existing 2.5.4 clients to lose functionality or behave differently than they did before (except for bugs that were fixed, of course). On the other hand, upgrading one of those clients to 2.6.0, along with the server, might make *new* functionality available for that client, functionality that 2.5.4 clients don't know how to take advantage of. If that happens, then the upgrade is *not* "forward-compatible": clearly you can't now downgrade that client back to 2.5.4 and keep all the functionality it had at 2.6.0, since some of that functionality was new in 2.6.0.

This is why micro releases are essentially for bug fixes only. They must remain compatible in both directions: if you upgrade from 2.5.3 to 2.5.4, then change your mind and downgrade back to 2.5.3, no functionality should be lost. Of course, the bugs fixed in 2.5.4 would reappear after the downgrade, but you wouldn't lose any features, except insofar as the restored bugs prevent the use of some existing features.

Client/server protocols are just one of many possible compatibility domains. Another is data formats: does the software write data to permanent storage? If so, the formats it reads and writes need to follow the compatibility guidelines promised by the release number policy. Version 2.6.0 needs to be able to read the files written by 2.5.4, but may silently upgrade the format to something that 2.5.4 cannot read, because the ability to downgrade is not required across a minor number boundary. If your project distributes code libraries for other programs to use, then APIs are a compatibility domain too: you must make sure that source and binary compatibility rules are spelled out in such a way that the informed user need never wonder whether or not it's safe to upgrade in place. She will be able to look at the numbers and know instantly.

In this system, you don't get a chance for a fresh start until you increment the major number. This can often be a real inconvenience: there may be features you wish to add, or protocols that you wish to redesign, that simply cannot be done while maintaining compatibility. There's no magic solution to this, except to try to design things in an extensible way in the first place (a topic easily worth its own book, and certainly outside the scope of this one). But publishing a release compatibility policy, and adhering to it, is an inescapable part of distributing software. One nasty surprise can alienate a lot of users. The policy just described is good partly because it's already quite widespread, but also because it's easy to explain and to remember, even for those not already familiar with it.

It is generally understood that these rules do not apply to pre-1.0 releases (although your release policy should probably state so explicitly, just to be clear). A project that is still in initial development can release 0.1, 0.2, 0.3, and so on in sequence, until it's ready for 1.0, and the differences between those releases can be arbitrarily large. Micro numbers in pre-1.0 releases are optional. Depending on the nature of your project and the differences between the releases, you might find it useful to have 0.1.0, 0.1.1, etc., or you might not. Conventions for pre-1.0 release numbers are fairly loose, mainly because people understand that strong compatibility constraints would hamper early development too much, and because early adopters tend to be forgiving anyway.

Remember that all these injunctions only apply to this particular three-component system. Your project could easily come up with a different three-component system, or even decide it doesn't need such fine granularity and use a two-component system instead. The important thing is to decide early, publish exactly what the components mean, and stick to it.

The Even/Odd Strategy

Some projects use the parity of the minor number component to indicate the stability of the software: even means stable, odd means unstable. This applies only to the minor number, not the major or micro numbers. Increments in the micro number still indicate bug fixes (no new features), and increments in the major number still indicate big changes, new feature sets, etc.

The advantage of the even/odd system, which has been used by the Linux kernel project[3] among others, is that it offers a way to release new functionality for testing without subjecting production users to potentially unstable code. People can see from the numbers that "2.4.21" is okay to install on their live web server, but that "2.5.1" should probably stay confined to home workstation experiments. The development team handles the bug reports that come in from the unstable (odd-minor-numbered) series, and when things start to settle down after some number of micro releases in that series, they increment the minor number (thus making it even), reset the micro number back to "0", and release a presumably stable package.

This system preserves, or at least, does not conflict with, the compatibility guidelines given earlier. It simply overloads the minor number with some extra information. This forces the minor number to be incremented about twice as often as would otherwise be necessary, but there's no real harm in that. The even/odd system is probably best for projects that have very long release cycles, and which by their nature have a high proportion of conservative users who value stability above new features. It is not the only way to get new functionality tested in the wild, however. In the section called “Stabilizing a Release” later in this chapter we will examine another, perhaps more common, method of releasing potentially unstable code to the public, in which the release number is further marked so that people have an idea of the risk/benefit trade-offs immediately on seeing the release's name.

Release Branches

From a developer's point of view, a free software project is in a state of continuous release. Developers usually run the latest available code at all times, because they want to spot bugs, and because they follow the project closely enough to be able to stay away from currently unstable areas of the feature space. They often update their copy of the software every day, sometimes more than once a day, and when they check in a change, they can reasonably expect that every other developer will have it within a day or two.

How, then, should the project make a formal release? Should it simply take a snapshot of the tree at a moment in time, package it up, and hand it to the world as, say, version "3.5.0"? Common sense says no. First, there may be no moment in time when the entire development tree is clean and ready for release. Newly-started features could be lying around in various states of completion. Someone might have checked in a major change to fix a bug, but the change could be controversial and under debate at the moment the snapshot is taken. If so, it wouldn't work to simply delay the snapshot until the debate ends, because another, unrelated debate could start in the meantime, and then you'd have wait for *that* one to end too. This process is not guaranteed to halt.

In any case, using full-tree snapshots for releases would interfere with ongoing development work, even if the tree could be put into a releasable state. Say this snapshot is going to be "3.5.0"; presumably, the next snapshot would be "3.5.1", and would contain mostly fixes for bugs found in the 3.5.0 release. But if both are snapshots from the same tree, what are the developers supposed to do in the time between the two releases? They can't be adding new features; the compatibility guidelines prevent that. But not everyone will be enthusiastic about fixing bugs in the 3.5.0 code. Some people may have new features they're trying to complete, and will become irate if they are forced to choose between sitting idle and working on things they're not interested in, just because the project's release processes demand that the development tree remain unnaturally quiescent.

The solution to these problems is to always use a *release branch*. A release branch is just a branch in the version control system (see *branch*), on which the code destined for this release can be isolated from mainline development. The concept of release branches is certainly not original to free software; many proprietary development organizations use them too. However, in closed-source environments, release

[3]Though Linux no longer uses it; see en.wikipedia.org/wiki/Linux_kernel#Version_numbering [https://en.wikipedia.org/wiki/Linux_kernel#Version_numbering].

branches are sometimes considered a luxury—a kind of theoretical "best practice" that can, in the heat of a major deadline, be dispensed with while everyone on the team scrambles to stabilize the main tree.

Release branches are pretty much required in open source projects, however. I have seen projects do releases without them, but it has always resulted in some developers sitting idle while others—usually a minority—work on getting the release out the door. The result is usually bad in several ways. First, overall development momentum is slowed. Second, the release is of poorer quality than it needed to be, because there were only a few people working on it, and they were hurrying to finish so everyone else could get back to work. Third, it divides the development team psychologically, by setting up a situation in which different types of work interfere with each other unnecessarily. The developers sitting idle would probably be happy to contribute *some* of their attention to a release branch, as long as that were a choice they could make according to their own schedules and interests. But without the branch, their choice becomes "Do I participate in the project today or not?" instead of "Do I work on the release today, or work on that new feature I've been developing in the mainline code?"

Mechanics of Release Branches

The exact mechanics of creating a release branch depend on your version control system, of course, but the general concepts are the same in most systems. A branch usually sprouts from another branch or from the trunk. Traditionally, the trunk is where mainline development goes on, unfettered by release constraints, and, say, the first release branch, the one leading to the "1.0" release, sprouts off the trunk. (The details of how to create and manage branches in your particular version control system are beyond the scope of this book, but the semantics are roughly the same everywhere.) Note that you might want to name the branch "1.0.x" (with a literal "x") instead of "1.0.0". That way you can use the same minor line—i.e., the same branch—for all the micro releases in that line.

The social and technical process of stabilizing the branch for release is covered in the section called “Stabilizing a Release” later in this chapter. Here we are concerned just with the high-level version control actions that relate tothe release process. When the release branch is stabilized and ready, it is time to tag a snapshot from the branch (see *tag or snapshot* in Chapter 3, *Technical Infrastructure*) with a name like, e.g., "1.0.0". The resultant tag represents the exact state of the project's source tree in the 1.0.0 release (this is useful when developers need to compare against an old version while tracking down a bug). The next micro release in the same line is likewise prepared on the 1.0.x branch, and when it is ready, a tag is made for 1.0.1. Lather, rinse, repeat for 1.0.2, and so on. When it's time to start thinking about a 1.1.x release, make a new branch from trunk.

Maintenance can continue in parallel along both 1.0.x and 1.1.x, and releases can be made independently from both lines (while regular development work happens, as always, on the main trunk — in Git, the "master" branch). In fact, it is not unusual to publish near-simultaneous releases from two different lines. The older series is recommended for more conservative site administrators, who may not want to make the big jump to (say) 1.1 without careful preparation. Meanwhile, more adventurous people usually take the most recent release on the highest line, to make sure they're getting the latest features, even at the risk of greater instability.

This is not the only release branch strategy, of course. In some circumstances it may not even be the best, though it's worked out pretty well for projects I've been involved in. Use any strategy that seems to work, but remember the main points: the purpose of a release branch is to isolate release work from the fluctuations of daily development, and to give the project a physical entity—the release branch—around which to organize its release process. That process is described in detail in the next section.

Stabilizing a Release

Stabilization is the process of getting a release branch into a releasable state; that is, of deciding which changes will be in the release, which will not, and shaping the branch content accordingly.

There's a lot of potential grief contained in the word "deciding". The last-minute feature rush is a familiar phenomenon in collaborative software projects: as soon as developers see that a release is about to happen, they scramble to finish their current changes, in order not to miss the boat. This, of course, is the exact opposite of what you want at release time. It would be much better for people to work on features at a comfortable pace, and not worry too much about whether their changes make it into this release or the next one. The more changes one tries to cram into a release at the last minute, the more the code is destabilized, and (usually) the more new bugs are created.

Time-based releases vs feature-based releases.

Some software projects use "*time-based releases*", as opposed to "*feature-based releases*". With time-based releases, the project puts out a new releases at an absolutely regular rhythm, typically something like every six months, and the rule is that the release goes out no matter what new features and bugfixes are ready or not ready — anything that isn't ready simply isn't included in the release.[4] Developers who didn't make the deadline are told to just wait for the next train, but this is easy for them to accept because they can count on the next train coming by in exactly six months (or whatever the release period is) anyway. The advice in this section applies to both time-based and feature-based releases, but keep both methods in mind as you read. Depending on your project's goals or culture, one or other other method may be more appropriate.

Most software engineers agree in theory on rough criteria for what changes should be allowed into a release line during its stabilization period. Obviously, fixes for severe bugs can go in, especially for bugs without workarounds. Documentation updates are fine, as are fixes to error messages (except when they are considered part of the interface and must remain stable). Many projects also allow certain kinds of low-risk or non-core changes to go in during stabilization, and may have formal guidelines for measuring risk. But no amount of formalization can obviate the need for human judgement. There will always be cases where the project simply has to make a decision about whether a given change can go into a release. The danger is that since each person wants to see their own favorite changes admitted into the release, then there will be plenty of people motivated to allow changes, and not enough people motivated to bar them.

Thus, the process of stabilizing a release is mostly about creating mechanisms for saying "no". The trick for open source projects, in particular, is to come up with ways of saying "no" that won't result in too many hurt feelings or disappointed developers, and also won't prevent deserving changes from getting into the release. There are many different ways to do this. It's pretty easy to design systems that satisfy these criteria, once the team has focused on them as the important criteria. Here I'll briefly describe two of the most popular systems, at the extreme ends of the spectrum, but don't let that discourage your project from being creative. Plenty of other arrangements are possible; these are just two that I've seen work in practice.

Dictatorship by Release Owner

The group agrees to let one person be the *release owner*. This person has final say over what changes make it into the release. Of course, it is normal and expected for there to be discussions and arguments, but in the end the group must grant the release owner sufficient authority to make final decisions. For this system to work, it is necessary to choose someone with the technical competence to understand all the changes, and the social standing and people skills to navigate the discussions leading up to the release without causing too many hurt feelings.

A common pattern is for the release owner to say "I don't think there's anything wrong with this change, but we haven't had enough time to test it yet, so it shouldn't go into this release." It helps a lot if the re-

[4]While any release methology requires some degree of branch management, time-based releases imply that the development team must use fairly strict branch management discipline at all times. Otherwise, unfinished code might be hard to extricate from the release branch when release time rolls around.)

lease owner has broad technical knowledge of the project, and can give reasons why the change could be potentially destabilizing (for example, its interactions with other parts of the software, or portability concerns). People will sometimes ask such decisions to be justified, or will argue that a change is not as risky as it looks. These conversations need not be confrontational, as long as the release owner is able to consider all the arguments objectively and not reflexively dig in her heels.

Note that the release owner need not be the same person as the project leader (in cases where there is a project leader at all; see the section called “Benevolent Dictators” in Chapter 4, *Social and Political Infrastructure*). In fact, sometimes it's good to make sure they're *not* the same person. The skills that make a good development leader are not necessarily the same as those that make a good release owner. In something as important as the release process, it may be wise to have someone provide a counterbalance to the project leader's judgement. In that case, the project leader needs to remember that overriding a decision by the release owner will undermine the release owner's authority; that alone may be enough reason, in most situations, to let the release owner win when there is a disagreement.

Contrast the release owner role with the less dictatorial role described in the section called “Release manager” later in this chapter.

Voting on Changes

At the opposite extreme from dictatorship by release owner, developers can simply vote on which changes to include in the release. However, since the most important function of release stabilization is to *exclude* changes, it's important to design the voting system in such a way that getting a change into the release involves positive action by multiple developers. Including a change should need more than just a simple majority (see the section called “Who Votes?” in Chapter 4, *Social and Political Infrastructure*). Otherwise, one vote for and none against a given change would suffice to get it into the release, and an unfortunate dynamic would be set up whereby each developer would vote for her own changes, yet would be reluctant to vote against others' changes, for fear of possible retaliation. To avoid this, the system should be arranged such that subgroups of developers must act in cooperation to get any change into the release. This not only means that more people review each change, it also makes any individual developer less hesitant to vote against a change, because she knows that no particular one among those who voted for it would take her vote against as a personal affront. The greater the number of people involved, the more the discussion becomes about the change and less about the individuals.

The system used for many years in the Subversion project seems to have struck a good balance, so I'll recommend it here. In order for a change to be applied to the release branch, at least three developers must vote in favor of it, and none against. A single "no" vote is enough to stop the change from being included; that is, a "no" vote in a release context is equivalent to a veto (see the section called “Vetoes”). Naturally, any such vote must be accompanied by a justification, and in theory the veto could be overridden if enough people feel it is unreasonable and force a special vote over it. In practice, this never happens. People are conservative around releases anyway, and when someone feels strongly enough to veto the inclusion of a change, there's usually a good reason for it.

Because the release procedure is deliberately biased toward conservatism, the justifications offered for vetoes are sometimes procedural rather than technical. For example, a person may feel that a change is well-written and unlikely to cause any new bugs, but vote against its inclusion in a micro release simply because it's too big—perhaps it adds a new feature, or in some subtle way fails to fully follow the compatibility guidelines. I've occasionally even seen developers veto something because they simply had a gut feeling that the change needed more testing, even though they couldn't spot any bugs in it by inspection. People grumbled a little bit, but the vetoes stood and the change was not included in the release (I don't remember if any bugs were found in later testing or not, though).

Managing collaborative release stabilization

If your project chooses a change voting system, it is imperative that the physical mechanics of setting up ballots and casting votes be as convenient as possible. Although there is plenty of open source electronic voting software available, in practice the easiest thing to do is just to set up a text file in the release branch, called `STATUS` or `VOTES` or something like that. This file lists each proposed change—any developer can propose a change for inclusion—along with all the votes for and against it, plus any notes or comments. (Proposing a change doesn't necessarily mean voting for it, by the way, although the two often go together.) An entry in such a file might look like this:

```
* commit b31910a7180fc (issue #49)
  Prevent client/server handshake from happening twice.
  Justification:
    Avoids extra network turnaround; small change and easy to review.
  Notes:
    This was discussed in http://.../mailing-lists/message-7777.html
    and other messages in that thread.
  Votes:
    +1: jsmith, kimf
    -1: tmartin (breaks compatibility with some pre-1.0 servers;
                 admittedly, those servers are buggy, but why be
                 incompatible if we don't have to?)
```

In this case, the change acquired two positive votes, but was vetoed by tmartin, who gave the reason for the veto in a parenthetical note. The exact format of the entry doesn't matter; whatever your project settles on is fine—perhaps tmartin's explanation for the veto should go up in the "Notes:" section, or perhaps the change description should get a "Description:" header to match the other sections. The important thing is that all the information needed to evaluate the change be easily accessible, and that the mechanism for casting votes be as lightweight as possible. The proposed change is referred to by its revision number in the repository (in the above case a single commit, b31910a7180fc, although a proposed change could just as easily consist of multiple commits). The revision is assumed to refer to a change made on the trunk; if the change were already on the release branch, there would be no need to vote on it. If your version control system doesn't have an obvious syntax for referring to individual changes, then the project should make one up. For voting to be practical, each change under consideration must be unambiguously identifiable.[5]

Those proposing or voting for a change are responsible for making sure it applies cleanly to the release branch, that is, applies without conflicts (see *conflict*). If there are conflicts, then the entry should either point to an adjusted patch that does apply cleanly, or better yet to a temporary branch that holds an adjusted version of the change, for example:

```
* r13222, r13223, r13232
  Rewrite libsvn_fs_fs's auto-merge algorithm
  Justification:
    unacceptable performance (>50 minutes for a small commit) in
    a repository with 300,000 revisions
  Branch:
    1.1.x-r13222@13517
  Votes:
    +1: epg, ghudson
```

[5]For projects in Git, a "merge request" or "pull request" is usually the right unit for uniquely identifying a change.

That example is taken from real life; it comes from the STATUS file for the Subversion 1.1.4 release process. Notice how it uses the original revisions as canonical handles on the change, even though there is also a branch with a conflict-adjusted version of the change (the branch also combines the three trunk revisions into one, r13517, to make it easier to merge the change into the release, should it get approval). The original revisions are provided because they're still the easiest entity to review, since they have the original log messages. The temporary branch wouldn't have those log messages; in order to avoid duplication of information (see the section called “Singularity of information” in Chapter 3, *Technical Infrastructure*), the branch's log message for r13517 should simply say "Adjust r13222, r13223, and r13232 for backport to 1.1.x branch." All other information about the changes can be chased down at their original revisions.

Release manager

The actual process of merging (see *merge or port*) approved changes into the release branch can be performed by any developer. There does not need to be one person whose job it is to merge changes; if there are a lot of changes, it can be better to spread the burden around.

However, although both voting and merging happen in a decentralized fashion, in practice there are usually one or two people driving the release process. This role is sometimes formally blessed as *release manager*, but it is quite different from a release owner (see the section called “Dictatorship by Release Owner” earlier in this chapter) who has final say over the changes. Release managers keep track of how many changes are currently under consideration, how many have been approved, how many seem likely to be approved, etc. If they sense that important changes are not getting enough attention, and might be left out of the release for lack of votes, they will gently nag other developers to review and vote. When a batch of changes are approved, these people will often take it upon themselves to merge them into the release branch; it's fine if others leave that task to them, as long as everyone understands that the release managers are not obligated to do all the work unless they have explicitly committed to it. When the time comes to put the release out the door (see the section called “Testing and Releasing” later in this chapter), the release managers also take care of the logistics of creating the final release packages, collecting digital signatures, uploading the packages, and making the public announcement.

Packaging

The canonical form for distribution of free software is as source code. This is true regardless of whether the software normally runs in source form (i.e., can be interpreted, like Perl, Python, PHP, etc.) or needs to be compiled first (like C, C++, Java, etc.). With compiled software, most users will probably not compile the sources themselves, but will instead install from pre-built binary packages (see the section called “Binary Packages” later in this chapter). However, those binary packages are still derived from a master source distribution. The point of the source package is to unambiguously define the release. When the project distributes "Scanley 2.5.0", what it means, specifically, is "The tree of source code files that, when compiled (if necessary) and installed, produces Scanley 2.5.0."

There is a fairly strict standard for how source releases should look. One will occasionally see deviations from this standard, but they are the exception, not the rule. Unless there is a compelling reason to do otherwise, your project should follow this standard too.

Format

The source code should be shipped in the standard formats for transporting directory trees. For Unix and Unix-like operating systems, the convention is to use TAR format, compressed by **compress**, **gzip**, **bzip** or **bzip2**. For MS Windows, the standard method for distributing directory trees is *zip* format, which

compresses automatically. For JavaScript projects, it is customary to ship the "minified"[6] versions of the files together with the human-readable source files.

Name and Layout

The name of the package should consist of the software's name plus the release number, plus the format suffixes appropriate for the archive type. For example, Scanley 2.5.0, packaged for Unix using GNU Zip (gzip) compression, would look like this:

scanley-2.5.0.tar.gz

or for Windows using zip compression:

scanley-2.5.0.zip

Either of these archives, when unpacked, should create a single new directory tree named `scanley-2.5.0` in the current directory. Underneath the new directory, the source code should be arranged in a layout ready for compilation (if compilation is needed) and installation. In the top level of new directory tree, there should be a plain text `README` file explaining what the software does and what release this is, and giving pointers to other resources, such as the project's web site, other files of interest, etc. Among those other files should be an `INSTALL` file, sibling to the `README` file, giving instructions on how to build and install the software for all the operating systems it supports. As mentioned in the section called “How to Apply a License to Your Software” in Chapter 2, *Getting Started*, there should also be a `COPYING` or `LICENSE` file, giving the software's terms of distribution.[7]

There should also be a `CHANGES` file (sometimes called `NEWS`), explaining what's new in this release. The `CHANGES` file accumulates changelists for all releases, in reverse chronological order, so that the list for this release appears at the top of the file. Completing that list is usually the last thing done on a stabilizing release branch; some projects write the list piecemeal as they're developing, others prefer to save it all up for the end and have one person write it, getting information by combing the version control logs. The list looks something like this:

```
Version 2.5.0
(20 December 2014, from branches 2.5.x)
http://scanley.org/repos/tags/2.5.0/

 New features, enhancements:
    * Added regular expression queries (issue #53)
    * Added support for UTF-8 and UTF-16 documents
    * Documentation translated into Polish, Russian, Malagasy
    * ...

 Bugfixes:
    * fixed reindexing bug (issue #945)
    * fixed some query bugs (issues #815, #1007, #1008)
    * ...
```

The list can be as long as necessary, but don't bother to include every little bugfix and feature enhancement. Its purpose is to give users an overview of what they would gain by upgrading to the new release, and to tell them about any incompatible changes. In fact, the changelist is customarily included in the

[6] See en.wikipedia.org/wiki/Minification_%28programming%29 [https://en.wikipedia.org/wiki/Minification_%28programming%29].
[7] Your all-caps files — README, INSTALL, etc — may of course have ".txt" extensions, or ".md" to indicate Markdown (daringfireball.net/projects/markdown [https://daringfireball.net/projects/markdown/]) format, etc.

announcement email (see the section called “Testing and Releasing” later in this chapter), so write it with that audience in mind.

The actual layout of the source code inside the tree should be the same as, or as similar as possible to, the source code layout one would get by checking out the project directly from its version control repository. Sometimes there are a few differences, for example because the package contains some generated files needed for configuration and compilation (see the section called “Compilation and Installation” later in this chapter), or because the distribution includes third-party software that is not maintained by the project, but that is required and that users are not likely to already have. But even if the distributed tree corresponds exactly to some development tree in the version control repository, the distribution itself should not be a working copy (see *working copy or working files*). The release is supposed to represent a static reference point—a particular, unchangeable configuration of source files. If it were a working copy, the danger would be that the user might update it, and afterward think that he still has the release when in fact he has something different.

Remember that the package is the same regardless of the packaging. The release—that is, the precise entity referred to when someone says "Scanley 2.5.0"—is the tree created by unpacking a zip file or tarball. So the project might offer all of these for download:

scanley-2.5.0.tar.bz2
scanley-2.5.0.tar.gz
scanley-2.5.0.zip

...but the source tree created by unpacking them would be the same. That source tree itself is the distribution; the form in which it is downloaded is merely a matter of convention or convenience. Certain minor differences between source packages are allowable: for example, in the Windows package, text files may have lines ending with CRLF (Carriage Return and Line Feed), while Unix packages would use just LF. The trees may be arranged slightly differently between source packages destined for different operating systems, too, if those operating systems require different sorts of layouts for compilation. However, these are all basically trivial transformations. The basic source files should be the same across all the packagings of a given release.

To capitalize or not to capitalize

When referring to a project by name, people generally capitalize it as a proper noun, and capitalize acronyms if there are any: "MySQL 5.0", "Scanley 2.5.0", etc. Whether this capitalization is reproduced in the package name is up to the project. Either `Scanley-2.5.0.tar.gz` or `scanley-2.5.0.tar.gz` would be fine, for example (I personally prefer the latter, because I don't like to make people hit the shift key, but plenty of projects ship capitalized packages). The important thing is that the directory created by unpacking the tarball use the same capitalization. There should be no surprises: the user must be able to predict with perfect accuracy the name of the directory that will be created when she unpacks a distribution.

Pre-releases

When shipping a pre-release or candidate release, the qualifier is a part of the release number, so include it in the name of the package's name. For example, the ordered sequence of alpha and beta releases given earlier in the section called “Release Number Components” would result in package names like this:

scanley-2.3.0-alpha1.tar.gz
scanley-2.3.0-alpha2.tar.gz
scanley-2.3.0-beta1.tar.gz
scanley-2.3.0-beta2.tar.gz
scanley-2.3.0-beta3.tar.gz
scanley-2.3.0.tar.gz

The first would unpack into a directory named `scanley-2.3.0-alpha1`, the second into `scanley-2.3.0-alpha2`, and so on.

Compilation and Installation

For software requiring compilation or installation from source, there are usually standard procedures that experienced users expect to be able to follow. For example, for programs written in C, C++, or certain other compiled languages, the standard under Unix-like systems is for the user to type:

```
$ ./configure
$ make
# make install
```

The first command autodetects as much about the environment as it can and prepares for the build process, the second command builds the software in place (but does not install it), and the last command installs it on the system. The first two commands are done as a regular user, the third as root. For more details about setting up this system, see the excellent *GNU Autoconf, Automake, and Libtool* book by Vaughan, Elliston, Tromey, and Taylor. It is published as treeware by New Riders, and its content is also freely available online at sources.redhat.com/autobook [http://sources.redhat.com/autobook/].

This is not the only standard, though it is one of the most widespread. Other programming languages often have their own standards for building and installing packages. If it's not obvious to you what the applicable standards are for your project, ask an experienced developer; you can safely assume that *some* standard applies, even if you don't know what it is at first.

Whatever the appropriate standards for you project are, don't deviate from them unless you absolutely must. Standard installation procedures are practically spinal reflexes for a lot of system administrators now. If they see familiar invocations documented in your project's `INSTALL` file, that instantly raises their faith that your project is generally aware of conventions, and that it is likely to have gotten other things right as well. Also, as discussed in the section called “Downloads” in Chapter 2, *Getting Started*, having a standard build procedure pleases potential developers.

On Windows, the standards for building and installing are a bit less settled. For projects requiring compilation, the general convention seems to be to ship a tree that can fit into the workspace/project model of the standard Microsoft development environments (Developer Studio, Visual Studio, VS.NET, MSVC++, etc.). Depending on the nature of your software, it may be possible to offer a Unix-like build option on Windows via the Cygwin (cygwin.com [https://www.cygwin.com/]) environment. And of course, if you're using a language or programming framework that comes with its own build and install conventions—e.g., Perl or Python—you should simply use whatever the standard method is for that framework, whether on Windows, Unix, Mac OS X, or any other operating system.

Be willing to put in a lot of extra effort in order to make your project conform to the relevant build or installation standards. Building and installing is an entry point: it's okay for things to get harder after that, if they absolutely must, but it would be a shame for the user's or developer's very first interaction with the software to require unexpected steps.

Binary Packages

Although the formal release is a source code package, users often install software from binary packages, either provided by their operating system's software distribution mechanism, or obtained manually from the project web site or from some third party. Here "binary" doesn't necessarily mean "compiled"; it's a general term for pre-configured form of the package that allows a user to install it on his computer without going through the usual source-based build and install procedures. On RedHat GNU/Linux, it is the RPM system; on Debian GNU/Linux, it is the APT (`.deb`) system; etc.

Whether these binary packages are assembled by people closely associated with the project, or by distant third parties, users are going to *treat* them as equivalent to the project's official releases, and will file tickets in the project's bug tracker based on the behavior of the binary packages. Therefore, it is in the project's interest to provide packagers with clear guidelines, and work closely with them to see to it that what they produce represents the software fairly and accurately.

The main thing packagers need to know is that they should always base their binary packages on an official source release. Sometimes packagers are tempted to pull an unstable incarnation of the code from the repository, or to include selected changes that were committed after the release was made, in order to provide users with certain bug fixes or other improvements. The packager thinks he is doing his users a favor by giving them the more recent code, but actually this practice can cause a great deal of confusion. Projects are prepared to receive reports of bugs found in released versions, and bugs found in recent trunk and major branch code (that is, found by people who deliberately run bleeding edge code). When a bug report comes in from these sources, the responder will often be able to confirm immediately that the bug is known to be present in that snapshot, and perhaps that it has since been fixed and that the user should upgrade or wait for the next release. If it is a previously unknown bug, knowing the precise release makes it easier to reproduce and easier to categorize in the tracker.

Projects are not prepared, however, to receive bug reports based on unspecified intermediate or hybrid versions. Such bugs can be hard to reproduce; also, they may be due to unexpected interactions between isolated changes pulled in from later development, and thereby cause misbehaviors that the project's developers should not have to take the blame for. I have even seen dismayingly large amounts of time wasted because a bug was *absent* when it should have been present: someone was running a slightly patched up version, based on (but not identical to) an official release, and when the predicted bug did not happen, everyone had to dig around a lot to figure out why.

Still, there will sometimes be circumstances when a packager insists that modifications to the source release are necessary[8]. Packagers should be encouraged to bring this up with the project's developers and describe their plans. They may get approval, but failing that, they will at least have notified the project of their intentions, so the project can watch out for unusual bug reports. The developers may respond by putting a disclaimer on the project's web site, and may ask that the packager do the same thing in the appropriate place, so that users of that binary package know what they are getting is not exactly the same as what the project officially released. There need be no animosity in such a situation, though unfortunately there often is. It's just that packagers have a slightly different set of goals from developers. The packagers mainly want the best out-of-the-box experience for their users. The developers want that too, of course, but they also need to ensure that they know what versions of the software are out there, so they can receive coherent bug reports and make compatibility guarantees. Sometimes these goals conflict. When they do, it's good to keep in mind that the project has no control over the packagers, and that the bonds of obligation run both ways. It's true that the project is doing the packagers a favor simply by producing the software. But the packagers are also doing the project a favor, by taking on a mostly unglamorous job in order to make the software more widely available, often by orders of magnitude. It's fine to disagree with packagers, but don't flame them; just try to work things out as best you can.

Testing and Releasing

Once the source distribution is produced from the stabilized release branch, the public part of the release process begins. But before the distribution is made available to the world at large, it should be tested and approved by some minimum number of developers, usually three or more. Approval is not simply a matter of inspecting the release for obvious flaws; ideally, the developers download the package, build and install it onto a clean system, run the regression test suite (see the section called “Automated testing” in Chapter 8, *Managing Participants*), and do some manual testing. Assuming it passes these checks, as

[8] en.wikipedia.org/wiki/Mozilla_Corporation_software_rebranded_by_the_Debian_project#Iceweasel [https://en.wikipedia.org/wiki/Mozilla_Corporation_software_rebranded_by_the_Debian_project#Iceweasel] gives a well-known example of this.

well as any other release checklist criteria the project may have, the developers then digitally sign each container (the .tar.gz file, .zip file, etc) using GnuPG (gnupg.org [https://www.gnupg.org/]), PGP (pgpi.org [http://www.pgpi.org/]), or some other program capable of producing PGP-compatible signatures.

In most projects, the developers just use their personal digital signatures, instead of a shared project key, and as many developers as want to may sign (i.e., there is a minimum number, but not a maximum). The more developers sign, the more testing the release undergoes, and also the greater the likelihood that a security-conscious user can find a digital trust path from herself to the release.

Once approved, the release (that is, all tarballs, zip files, and whatever other formats are being distributed) should be placed into the project's download area, accompanied by the digital signatures, and by MD5/SHA1 checksums (see en.wikipedia.org/wiki/Cryptographic_hash_function [https://en.wikipedia.org/wiki/Cryptographic_hash_function]). There are various standards for doing this. One way is to accompany each released package with a file giving the corresponding digital signatures, and another file giving the checksum. For example, if one of the released packages is `scanley-2.5.0.tar.gz`, place in the same directory a file `scanley-2.5.0.tar.gz.asc` containing the digital signature for that tarball, another file `scanley-2.5.0.tar.gz.md5` containing its MD5 checksum, and optionally another, `scanley-2.5.0.tar.gz.sha1`, containing the SHA1 checksum. A different way to provide checking is to collect all the signatures for all the released packages into a single file, `scanley-2.5.0.sigs`; the same may be done with the checksums.

It doesn't really matter which way you do it. Just keep to a simple scheme, describe it clearly, and be consistent from release to release. The purpose of all this signing and checksumming is to give users a way to verify that the copy they receive has not been maliciously tampered with. Users are about to run this code on their computers—if the code has been tampered with, an attacker could suddenly have a back door to all their data. See the section called “Security Releases” later in this chapter for more about paranoia.

Candidate Releases

For important releases containing many changes, many projects prefer to put out *release candidates* first, e.g., `scanley-2.5.0-beta1` before `scanley-2.5.0`. The purpose of a candidate is to subject the code to wide testing before blessing it as an official release. If problems are found, they are fixed on the release branch and a new candidate release is rolled out (`scanley-2.5.0-beta2`). The cycle continues until no unacceptable bugs are left, at which point the last candidate release becomes the official release—that is, the only difference between the last candidate release and the real release is the removal of the qualifier from the version number.

In most other respects, a candidate release should be treated the same as a real release. The *alpha*, *beta*, or *rc* qualifier is enough to warn conservative users to wait until the real release, and of course the announcement emails for the candidate releases should point out that their purpose is to solicit feedback. Other than that, give candidate releases the same amount of care as regular releases. After all, you want people to use the candidates, because exposure is the best way to uncover bugs, and also because you never know which candidate release will end up becoming the official release.

Announcing Releases

Announcing a release is like announcing any other event, and should use the procedures described in the section called “Publicity” in Chapter 6, *Communications*. There are a few specific things to do for releases, though.

Whenever you write the URL to the downloadable release tarball, make sure to also write the MD5/SHA1 checksums and pointers to the digital signatures file. Since the announcement happens in multiple forums (mailing list, news page, etc.), this means users can get the checksums from multiple sources, which gives the most security-conscious among them extra assurance that the checksums themselves

have not been tampered with. Meanwhile, giving the link to the digital signature files multiple times doesn't make those signatures more secure, but it does reassure people (especially those who don't follow the project closely) that the project takes security seriously.

In the announcement email, and on news pages that contain more than just a blurb about the release, make sure to include the relevant portion of the CHANGES file, so people can see why it might be in their interests to upgrade. This is as important with candidate releases as with final releases; the presence of bugfixes and new features is important in tempting people to try out a candidate release.

Finally, don't forget to thank the development team, the testers, and all the people who took the time to file good bug reports. Don't single out anyone by name, though, unless there's someone who is individually responsible for a huge piece of work, the value of which is widely recognized by everyone in the project. Be wary of sliding down the slippery slope of credit inflation (see the section called “Credit” in Chapter 8, *Managing Participants*).

Maintaining Multiple Release Lines

Most mature projects maintain multiple release lines in parallel. For example, after 1.0.0 comes out, that line should continue with micro (bugfix) releases 1.0.1, 1.0.2, etc., until the project explicitly decides to end the line. Note that merely releasing 1.1.0 is not sufficient reason to end the 1.0.x line. For example, some users make it a policy never to upgrade to the first release in a new minor or major series—they let others shake the bugs out of, say 1.1.0, and wait until 1.1.1. This isn't necessarily selfish (remember, they're forgoing the bugfixes and new features too); it's just that, for whatever reason, they've decided to be very careful with upgrades. Accordingly, if the project learns of a major bug in 1.0.3 right before it's about to release 1.1.0, it would be a bit severe to just put the bugfix in 1.1.0 and tell all the old 1.0.x users they should upgrade. Why not release both 1.1.0 and 1.0.4, so everyone can be happy?

After the 1.1.x line is well under way, you can declare 1.0.x to be at *end of life*. This should be announced officially. The announcement could stand alone, or it could be mentioned as part of a 1.1.x release announcement; however you do it, users need to know that the old line is being phased out, so they can make upgrade decisions accordingly.

Some projects set a window of time during which they pledge to support the previous release line. In an open source context, "support" means accepting bug reports against that line, and making maintenance releases when significant bugs are found. Other projects don't give a definite amount of time, but watch incoming bug reports to gauge how many people are still using the older line. When the percentage drops below a certain point, they declare end of life for the line and stop supporting it.

For each release, make sure to have a *target version* or *target milestone* available in the bug tracker, so people filing bugs will be able to do so against the proper release. Don't forget to also have a target called "development" or "latest" for the most recent development sources, since some people—not only active developers—will often stay ahead of the official releases.

Security Releases

Most of the details of handling security bugs were covered in the section called “Announcing Security Vulnerabilities” in Chapter 6, *Communications*, but there are some special details to discuss for doing security releases.

A *security release* is a release made solely to close a security vulnerability. The code that fixes the bug cannot be made public until the release is available, which means not only that the fixes cannot be committed to the repository until the day of the release, but also that the release cannot be publicly tested before it goes out the door. Obviously, the developers can examine the fix among themselves, and test the release privately, but widespread real-world testing is not possible.

Because of this lack of testing, a security release should always consist of some existing release plus the fixes for the security bug, with *no other changes*. This is because the more changes you ship without testing, the more likely that one of them will cause a new bug, perhaps even a new security bug! This conservatism is also friendly to administrators who may need to deploy the security fix, but whose upgrade policy stipulates that they not deploy any other changes at the same time.

Making a security release sometimes involves some minor deception. For example, the project may have been working on a 1.1.3 release, with certain bug fixes to 1.1.2 already publicly declared, when a security report comes in. Naturally, the developers cannot talk about the security problem until they make the fix available; until then, they must continue to talk publicly as though 1.1.3 will be what it's always been planned to be. But when 1.1.3 actually comes out, it will differ from 1.1.2 only in the security fixes, and all those other fixes will have been deferred to 1.1.4 (which, of course, will now *also* contain the security fix, as will all other future releases).

You could add an extra component to an existing release to indicate that it contains security changes only. For example, people would be able to tell just from the numbers that 1.1.2.1 is a security release against 1.1.2, and they would know that any release "higher" than that (e.g., 1.1.3, 1.2.0, etc.) contains the same security fixes. For those in the know, this system conveys a lot of information. On the other hand, for those not following the project closely, it can be a bit confusing to see a three-component release number most of the time with an occasional four-component one thrown in seemingly at random. Most projects I've looked at choose consistency and simply use the next regularly scheduled number for security releases, even when it means shifting other planned releases by one number.

Releases and Daily Development

Maintaining parallel releases simultaneously has implications for how daily development is done. In particular, it makes practically mandatory a discipline that would be recommended anyway: have each commit be a single logical change, and don't mix unrelated changes in the same commit. If a change is too big or too disruptive to do in one commit, break it across N commits, where each commit is a well-partitioned subset of the overall change, and includes nothing unrelated to the overall change.

Here's an example of an ill-thought-out commit:

```
commit 3b1917a01f8c50e25db0b71edce32357d2645759
Author: J. Random <jrandom@example.com>
Date:   Sat 2014-06-28 15:53:07 -0500

Fix Issue #1729: warn on change during re-indexing.

Make indexing gracefully warn the user when a file is changing as it
is being indexed.

* ui/repl.py
  (ChangingFile): New exception class.
  (DoIndex): Handle new exception.

* indexer/index.py
  (FollowStream): Raise new exception if file changes during indexing.
  (BuildDir): Unrelatedly, remove some obsolete comments, reformat
  some code, and fix the error check when creating a directory.

Other unrelated cleanups:
```

```
* www/index.html: Fix some typos, set next release date.
------------------------------------------------------------------------
```

The problem with it becomes apparent as soon as someone needs to port the `BuildDir` error check fix over to a branch for an upcoming maintenance release. The porter doesn't want any of the other changes—for example, perhaps the fix for ticket #1729 wasn't approved for the maintenance branch at all, and the `index.html` tweaks would simply be irrelevant there. But she cannot easily grab just the `BuildDir` change via the version control tool's merge functionality, because the version control system was told that that change is logically grouped with all these other unrelated things. In fact, the problem would become apparent even before the merge. Merely listing the change for voting would become problematic: instead of just giving the revision number, the proposer would have to make a special patch or change branch just to isolate the portion of the commit being proposed. That would be a lot of work for others to suffer through, and all because the original committer couldn't be bothered to break things into logical groups.

In fact, that commit really should have been *four* separate commits: one to fix issue #1729, another to remove obsolete comments and reformat code in `BuildDir`, another to fix the error check in `BuildDir`, and finally, one to tweak `index.html`. The third of those commits would be the one proposed for the maintenance release branch.

Of course, release stabilization is not the only reason why having each commit be one logical change is desirable. Psychologically, a semantically unified commit is easier to review, and easier to revert if necessary (in some version control systems, reversion is really a special kind of merge anyway). A little upfront discipline on each developer's part can save the project a lot of headache later.

Planning Releases

One area where open source projects have historically differed from proprietary projects is in release planning. Proprietary projects usually have firmer deadlines. Sometimes it's because customers were promised that an upgrade would be available by a certain date, because the new release needs to be coordinated with some other effort for marketing purposes, or because the venture capitalists who invested in the whole thing need to see some results before they put in any more funding. Free software projects, on the other hand, are concerned with maintaining a cooperative working atmosphere among many parties, some of them possibly business competitors, and the preservation of the working relationship can trump and single party's deadlines.

Of course, many open source projects are funded by corporations, and are correspondingly by deadline-conscious management. This is in many ways a good thing, but it can cause conflicts between the priorities of those developers who are being paid and those who are volunteering their time. These conflicts often happen around the issue of when and how to schedule releases. The salaried developers who are under pressure will naturally want to just pick a date when the releases will occur, and have everyone's activities fall into line. But the volunteers may have other agendas—perhaps features they want to complete, or some testing they want to have done—that they feel the release should wait on.

There is no general solution to this problem except discussion and compromise, of course. But you can minimize the frequency and degree of friction caused, by decoupling the proposed *existence* of a given release from the date when it would go out the door. That is, try to steer discussion toward the subject of which releases the project will be making in the near- to medium-term future, and what features will be in them, without at first mentioning anything about dates, except for rough guesses with wide margins of error[9]. By nailing down feature sets early, you reduce the complexity of the discussion centered

[9]For an alternative approach, you may wish to read Martin Michlmayr's Ph.D. thesis *Quality Improvement in Volunteer Free and Open Source Software Projects: Exploring the Impact of Release Management* (cyrius.com/publications/michlmayr-phd.html [https://www.cyrius.com/publications/michlmayr-phd.html]). It is about using time-based release processes, as opposed to feature-based, in large free software projects. See also Release Management in Free Software Projects: Practices and Problems [https://www.cyrius.com/publications/michlmayr_hunt_probert-re-

on any individual release, and therefore improve predictability. This also creates a kind of inertial bias against anyone who proposes to expand the definition of a release by adding new features or other complications. If the release's contents are fairly well defined, the onus is on the proposer to justify the expansion, even though the date of the release may not have been set yet.

In his multi-volume biography of Thomas Jefferson, *Jefferson and His Time*, Dumas Malone tells the story of how Jefferson handled the first meeting held to decide the organization of the future University of Virginia. The University had been Jefferson's idea in the first place, but (as is the case everywhere, not just in open source projects) many other parties had climbed on board quickly, each with their own interests and agendas. When they gathered at that first meeting to hash things out, Jefferson made sure to show up with meticulously prepared architectural drawings, detailed budgets for construction and operation, a proposed curriculum, and the names of specific faculty he wanted to import from Europe. No one else in the room was even remotely as prepared; the group essentially had to capitulate to Jefferson's vision, and the University was eventually founded more or less in accordance with his plans. The facts that construction went far over budget, and that many of his ideas did not, for various reasons, work out in the end, were all things Jefferson probably knew perfectly well would happen. His purpose was strategic: to show up at the meeting with something so substantive that everyone else would have to fall into the role of simply proposing modifications to it, so that the overall shape, and therefore schedule, of the project would be roughly as he wanted.

In the case of a free software project, there is no single "meeting", but instead a series of small proposals made in the discussion forums and in the ticket tracker. But if you have some credibility in the project to start with, and you start assigning various features, enhancements, and bugs to target releases in the tracker, according to some announced overall plan, people will mostly go along with you. Once you've got things laid out more or less as you want them, the conversations about actual release *dates* will go much more smoothly.

It is crucial, of course, to never present any individual decision as written in stone. In the comments associated with each assignment of a ticket to a specific future release, invite discussion, dissent, and be genuinely willing to be persuaded whenever possible. Never exercise control merely for the sake of exercising control: the more deeply others feel they can participate in the release planning process (see the section called “Share Management Tasks as Well as Technical Tasks” in Chapter 8, *Managing Participants*), the easier it will be to persuade them to share your priorities on the issues that really count for you.

The other way the project can lower tensions around release planning is to make releases fairly often. When there's a long time between releases, the importance of any individual release is magnified in everyone's minds; people are that much more crushed when their code doesn't make it in, because they know how long it might be until the next chance. Depending on the complexity of the release process and the nature of your project, somewhere between every three and six months is usually about the right gap between releases, though maintenance lines may put out micro releases a bit faster, if there is demand for them.

lease_management.pdf], by Martin Michlmayr, Francis Hunt, and David Probert. Finally, Michlmayr gave a talk at Google on the subject: Release Management in Large Free Software Projects [https://www.youtube.com/watch?v=IKsQsxubuAA].

Chapter 8. Managing Participants

Getting people to agree on what a project needs, and to work together to achieve it, requires more than just a genial atmosphere and a lack of obvious dysfunction. It requires someone, or several someones, consciously managing all the people involved. Managing participants who work for different organizations, or in some cases for themselves, may not be a technical craft in the same sense as computer programming, but it is a craft in the sense that it can be improved through study and practice.

This chapter is a grab-bag of specific techniques for managing diverse participants in an open source project. It draws, perhaps more heavily than previous chapters, on the Subversion project as a case study, partly because I was working on that project as I wrote the first edition of this book and had all the primary sources close at hand, and partly because it's more acceptable to cast critical stones into one's own glass house than into others'. But I have also seen in various other projects the benefits of applying—and the consequences of not applying—the recommendations that follow; when it is politically feasible to give examples from some of those other projects, I will do so.

Speaking of politics, this is as good a time as any to drag that much-maligned word out for a closer look. Many engineers like to think of politics as something other people engage in. "*I'm* just advocating the best course for the project, but *he's* raising objections for political reasons." I believe this distaste for politics (or for what is imagined to be politics) is especially strong in engineers because engineers are bought into the idea that some solutions are objectively superior to others. Thus, when someone acts in a way that seems motivated by outside considerations—say, the maintenance of his own position of influence, the lessening of someone else's influence, outright horse-trading, or avoiding hurting someone's feelings—other participants in the project may get annoyed. Of course, this rarely prevents them from behaving in the same way when their own vital interests are at stake.

If you consider "politics" a dirty word, and hope to keep your project free of it, give up right now. Politics are inevitable whenever people have to cooperatively manage a shared resource. In the case of an open source project, even though the code itself is not a shared resource (since it can be copied by anyone), attention, credibility, and influence in the project are very much shared resources: they are by definition not copyable, not forkable.

Thus it is quite reasonable that one of the considerations going into each person's decision-making process is the question of how a given action might affect his own future influence in the project. After all, if you trust your own judgement and skills, as most programmers do, then the potential loss of future influence has to be considered a technical result, in a sense. Similar reasoning applies to other behaviors that might seem, on their face, like "pure" politics. In fact, there is no such thing as pure politics: it is precisely because actions have multiple real-world consequences that people become politically conscious in the first place. Politics is, in the end, simply an acknowledgment that *all* consequences of decisions must be taken into account. If a particular decision leads to a result that most participants find technically satisfying, but involves a change in power relationships that leaves key people feeling isolated, the latter is just as important a result as the former. To ignore it would not be high-minded, but shortsighted.

So as you read the advice that follows, and as you work with your own project, remember that there is *no one* who is above politics. Appearing to be above politics is merely one particular political strategy, and sometimes a very useful one, but it is never the reality. Politics is simply what happens when people disagree on the use or allocation of a shared asset, and successful projects are those that evolve political mechanisms for managing such disagreement constructively.

Community and Motivation

Why do people work on free software projects?[1] Of course, in some cases the answer is that it's their job — their manager asked them to. But even then, most participants have some degree of intrinsic motivation that goes beyond a mere management request. As every manager knows, people are much more successful when they have their own motivations for wanting to succeed than when they are merely fulfilling management requests in return for a paycheck. Most open source developers — I would even go so far as to say the vast majority of them — are not in it only for the paycheck. There is something more to it than that.

When asked, many claim they do it because they want to produce good software, or want to be personally involved in fixing the bugs that matter to them. But these reasons are usually not the whole story. After all, could you imagine a participant staying with a project even if no one ever said a word in appreciation of her work, or listened to her in discussions? Of course not. Clearly, people spend time on free software for reasons beyond just an abstract desire to produce good code. Understanding people's true motivations will help you arrange things so as to attract and keep them. The desire to produce good software may be among those motivations, along with the challenge and educational value of working on hard problems. But humans also have a built-in desire to work with other humans, and to give and earn respect through cooperative activities. Groups engaged in cooperative activities must evolve norms of behavior such that status is acquired and kept through actions that help the group's goals.

Those norms won't always arise by themselves. For example, on some projects—experienced open source developers can probably name several off the tops of their heads—people apparently feel that status is acquired by posting frequently and verbosely. They don't come to this conclusion accidentally; they come to it because they are rewarded with respect for making long, intricate arguments, whether or not that actually helps the project. Following are some techniques for creating an atmosphere in which status-acquiring actions are also *constructive* actions.

Delegation

Delegation is not merely a way to spread the workload around; it is also a political and social tool. Consider all the effects when you ask someone to do something. The most obvious effect is that, if he accepts, he does the task and you don't. But another effect is that he is made aware that you trusted him to handle the task. Furthermore, if you made the request in a public forum, then he knows that others in the group have been made aware of that trust too. He may also feel some pressure to accept, which means you must ask in a way that allows him to decline gracefully if he doesn't really want the job. If the task requires coordination with others in the project, you are effectively proposing that he become more involved, form bonds that might not otherwise have been formed, and perhaps become a source of authority in some subdomain of the project. The added involvement may be daunting, or it may lead him to become engaged in other ways as well, from an increased feeling of overall commitment.

Because of all these effects, it often makes sense to ask someone else to do something even when you know you could do it faster or better yourself. Of course, there is sometimes a strict economic efficiency argument for this anyway: perhaps the opportunity cost of doing it yourself would be too high—there might be something even more important you could do with that time. But even when that kind of comparative advantage argument doesn't apply, you may *still* want to ask someone else to take on the task, because in the long run you want to draw that person deeper into the project, even if it means spending extra time watching over them at first. The converse technique also applies: if you occasionally volunteer for work that someone else doesn't want or have time to do, you will gain her good will and respect. Delegation and substitution are not just about getting individual tasks done; they're also about drawing people into a closer committment to the project.

[1] This question was studied in detail, with interesting results, in a paper by Karim Lakhani and Robert G. Wolf, entitled *Why Hackers Do What They Do: Understanding Motivation and Effort in Free/Open Source Software Projects*. See flosshub.org/node/53 [http://flosshub.org/node/53].

Distinguish clearly between inquiry and assignment

Sometimes it is fair to expect that a person will accept a particular task. For example, if someone writes a bug into the code, or commits code that fails to comply with project guidelines in some obvious way, then it is enough to point out the problem and thereafter behave as though you assume the person will take care of it. Also, if they have stated publicly that they will do something, it is reasonable to depend on that. But there are other situations where it is by no means clear that you have a right to expect action. The person may do as you ask, or may not. Since no one likes to be taken for granted, you need to be sensitive to the difference between these two types of situations, and tailor your requests accordingly.

One thing that almost always causes people instant annoyance is being asked to do something in a way that implies that you think it is clearly their responsibility to do it, when they feel otherwise. For example, assignment of incoming tickets is particularly fertile ground for this kind of annoyance. The participants in a project usually know who is expert in what areas, so when a bug report comes in, there will often be one or two people whom everyone knows could probably fix it quickly. However, if you assign the ticket over to one of those people without her prior permission, she may feel she has been put into an uncomfortable position. She senses the pressure of expectation, but also may feel that she is, in effect, being punished for her expertise. After all, the way one acquires expertise is by fixing bugs, so perhaps someone else should take this one! (Note that ticket trackers that automatically assign tickets to particular people based on information in the bug report are less likely to offend, because everyone knows that the assignment was made by an automated process, and is not an indication of human expectations.)

While it would be nice to spread the load as evenly as possible, there are certain times when you just want to encourage the person who can fix a bug the fastest to do so. Given that you can't afford a communications turnaround for every such assignment ("Would you be willing to look at this bug?" "Yes." "Okay, I'm assigning the ticket over to you then." "Okay."), you should simply make the assignment in the form of an inquiry, conveying no pressure. Virtually all ticket trackers allow a comment to be associated with the assignment of an ticket. In that comment, you can say something like this:

> Assigning this over to you, jrandom, because you're most familiar with this code. Feel free to bounce this back if you don't have time to look at it, though. (And let me know if you'd prefer not to receive such requests in the future.)

This distinguishes clearly between the *request* for assignment and the recipient's *acceptance* of that assignment. The audience here isn't only the assignee, it's everyone: the entire group sees a public confirmation of the assignee's expertise, but the message also makes it clear that the assignee is free to accept or decline the responsibility.

Follow up after you delegate

When you ask someone to do something, remember that you have done so, and follow up with him no matter what. Most requests are made in public forums, and are roughly of the form "Can you take care of X? Let us know either way; no problem if you can't, just need to know." You may or may not get a response. If you do, and the response is negative, the loop is closed—you'll need to try some other strategy for dealing with X. If there is a positive response, then keep an eye out for progress on the issue, and comment on the progress you do or don't see (everyone works better when they know someone else is appreciating their work). If there is no response after a few days, ask again, or post saying that you got no response and are looking for someone else to do it. Or just do it yourself, but still make sure to say that you got no response to the initial inquiry.

The purpose of publicly noting the lack of response is *not* to humiliate the person, and your remarks should be phrased so as not to have that effect. The purpose is simply to show that you keep track of what you have asked for, and that you notice the reactions you get. This makes people more likely to say yes next time, because they will observe (even if only unconsciously) that you are likely to notice any work they do, given that you noticed the much less visible event of someone failing to respond.

Notice what people are interested in

Another thing that makes people happy is to have their interests noticed—in general, the more aspects of someone's personality you notice and remember, the more comfortable he will be, and the more he will want to work with groups of which you are a part.

For example, there was a sharp distinction in the Subversion project between people who wanted to reach a definitive 1.0 release (which we eventually did), and people who mainly wanted to add new features and work on interesting problems but who didn't much care when 1.0 came out. Neither of these positions is better or worse than the other; they're just two different kinds of developers, and both kinds do lots of work on the project. But we swiftly learned that it was important to *not* assume that the excitement of the 1.0 drive was shared by everyone. Electronic media can be very deceptive: you may sense an atmosphere of shared purpose when, in fact, it's shared only by the people you happen to have been talking to, while others have completely different priorities.

The more aware you are of what different people want out of the project, the more effectively you can make requests of them. Even just demonstrating an understanding of what they want, without making any associated request, is useful, in that it confirms to each person that she's not just another particle in an undifferentiated mass.

Praise and Criticism

Praise and criticism are not opposites; in many ways, they are very similar. Both are primarily forms of attention, and are most effective when specific rather than generic. Both should be deployed with concrete goals in mind. Both can be diluted by inflation: praise too much or too often and you will devalue your praise; the same is true for criticism, though in practice, criticism is usually reactive and therefore a bit more resistant to devaluation.

An important feature of technical culture is that detailed, dispassionate criticism is often taken as a kind of praise (as discussed in the section called “Recognizing Rudeness” in Chapter 6, *Communications*), because of the implication that the recipient's work is worth the time required to analyze it. However, both of those conditions—*detailed* and *dispassionate*—must be met for this to be true. For example, if someone makes a sloppy change to the code, it is useless (and actually harmful) to follow up saying simply "That was sloppy." Sloppiness is ultimately a characteristic of a *person*, not of their work, and it's important to keep your reactions focused on the work. It's much more effective to describe all the things wrong with the change, tactfully and without malice. If this is the third or fourth careless change in a row by the same person, it's appropriate to say that—again without anger—at the end of your critique, to make it clear that the pattern has been noticed.

If someone does not improve in response to criticism, the solution is not more or stronger criticism. The solution is for the group to remove that person from the position of incompetence, in a way that minimizes hurt feelings as much as possible; see the section called “Transitions” later in this chapter for examples. That is a rare occurrence, however. Most people respond pretty well to criticism that is specific, detailed, and contains a clear (even if unspoken) expectation of improvement.

Praise won't hurt anyone's feelings, of course, but that doesn't mean it should be used any less carefully than criticism. Praise is a tool: before you use it, ask yourself *why* you want to use it. As a rule, it's not a good idea to regularly praise people for doing what they usually do, or for actions that are a normal and expected part of participating in the group. If you were to do that, it would be hard to know when to stop: should you praise *everyone* for doing the usual things? After all, if you leave some people out, they'll wonder why. It's much better to express praise and gratitude sparingly, in response to unusual or unexpected efforts, with the intention of encouraging more such efforts. When a participant seems to have moved permanently into a state of higher productivity, adjust your praise threshold for that person accordingly. Repeated praise for normal behavior gradually becomes meaningless anyway. Instead, that

person should sense that her high level of productivity is now considered normal and natural, and only work that goes beyond that should be specially noticed.

This is not to say that the person's contributions shouldn't be acknowledged, of course. But remember that if the project is set up right, everything that person does is already visible anyway, and so the group will know (and the person will know that the rest of the group knows) everything she does. There are also ways to acknowledge someone's work by means other than direct praise. You could mention in passing, while discussing a related topic, that she has done a lot of work in the given area and is the resident expert there; you could publicly consult her on some question about the code; or perhaps most effectively, you could conspicuously make further use of the work she has done, so she sees that others are now comfortable relying on the results of her work. It's probably not necessary to do these things in any calculated way. Someone who regularly makes large contributions in a project will know it, and will occupy a position of influence by default. There's usually no need to take explicit steps to ensure this, unless you sense that, for whatever reason, a contributor is underappreciated.

Prevent Territoriality

Watch out for participants who try to stake out exclusive ownership of certain areas of the project, and who seem to want to do all the work in those areas, to the extent of aggressively taking over work that others start. Such behavior may even seem healthy at first. After all, on the surface it looks like the person is taking on more responsibility, and showing increased activity within a given area. But in the long run, it is destructive. When people sense a "no trespassing" sign, they stay away. This results in reduced review in that area, and greater fragility, because the lone developer becomes a single point of failure. Worse, it fractures the cooperative, egalitarian spirit of the project. The theory should always be that any developer is welcome to help out on any task at any time. Of course, in practice things work a bit differently: people do have areas where they are more and less influential, and non-experts frequently defer to experts in certain domains of the project. But the key is that this is all voluntary: informal authority is granted based on competence and proven judgement, but it should never be actively *taken*. Even if the person desiring the authority really is competent, it is still crucial that she hold that authority informally, through the consensus of the group, that the exact boundaries of the authority remain fuzzy and subjective, and that the authority never cause her to exclude others from working in that area.

Rejecting or editing someone's work for technical reasons is an entirely different matter, of course. There, the decisive factor is the content of the work, not who happened to act as gatekeeper. It may be that the same person happens to do most of the reviewing for a given area, but as long as he never tries to prevent someone else from doing that work too, things are probably okay.

Cookie Licking

The wonderful term *cookie licking*, which I first heard from Sumana Harihareswara, can be used for the situation where someone claims, in front of the group, that they're going to take care of a certain task but then does nothing with it. As Sumana says[2]: "Nobody in their right mind would pick up and eat the licked cookie or finish the [task]." If you think you see an instance of cookie licking happening in your project, simply pointing it out may be enough to de-territorialize the task in question and make others consider picking it up (may be enough to sterilize the cookie, I guess, though at this point staying with the analogy may be more confusing than helpful).

In order to combat incipient territorialism, or even the appearance of it, many projects have taken the step of banning the inclusion of author names or designated maintainer names in source files. I wholeheartedly agree with this practice: we follow it in the Subversion project, and it is more or less official policy at the Apache Software Foundation. ASF member Sander Striker puts it this way:

[2]See opensourcebridge.org/sessions/1132 [http://opensourcebridge.org/sessions/1132].

> *At the Apache Software foundation we discourage the use of author tags in source code. There are various reasons for this, apart from the legal ramifications. Collaborative development is about working on projects as a group and caring for the project as a group. Giving credit is good, and should be done, but in a way that does not allow for false attribution, even by implication. There is no clear line for when to add or remove an author tag. Do you add your name when you change a comment? When you put in a one-line fix? Do you remove other author tags when you refactor the code and it looks 95% different? What do you do about people who go about touching every file, changing just enough to make the virtual author tag quota, so that their name will be everywhere?*
>
> *There are better ways to give credit, and our preference is to use those. From a technical standpoint author tags are unnecessary; if you wish to find out who wrote a particular piece of code, the version control system can be consulted to figure that out. Author tags also tend to get out of date. Do you really wish to be contacted in private about a piece of code you wrote five years ago and were glad to have forgotten?*

A software project's source code files are the core of its identity. They should reflect the fact that the developer community as a whole is responsible for them, and not be divided up into little fiefdoms.

People sometimes argue in favor of author or maintainer tags in source files on the grounds that this gives visible credit to those who have done the most work there. There are two problems with this argument. First, the tags inevitably raise the awkward question of how much work one must do to get one's own name listed there too. Second, they conflate the issue of credit with that of authority: having done work in the past does not imply ownership of the area where the work was done, but it's difficult if not impossible to avoid such an implication when individual names are listed at the tops of source files. In any case, credit information can already be obtained from the version control logs and other out-of-band mechanisms like mailing list archives, so no information is lost by banning it from the source files themselves.[3]

If your project decides to ban individual names from source files, make sure not to go overboard. For instance, many projects have a `contrib/` area where small tools and helper scripts are kept, often written by people who are otherwise not associated with the project. It's fine for those files to contain author names, because they are not really maintained by the project as a whole. On the other hand, if a contributed tool starts getting hacked on by other people in the project, eventually you may want to move it to a less isolated location and, assuming the original author approves, remove the author's name, so that the code looks like any other community-maintained resource. If the author is sensitive about this, compromise solutions are acceptable, for example:

```
# indexclean.py: Remove old data from a Scanley index.
#
# Original Author: K. Maru <kobayashi@yetanotheremailservice.com>
# Now Maintained By: The Scanley Project <http://www.scanley.org/>
#                    and K. Maru.
#
# ...
```

But it's better to avoid such compromises, if possible, and most authors are willing to be persuaded, because they're happy that their contribution is being made a more integral part of the project.

[3] But see the mailing list thread entitled *"having authors names in .py files"* at groups.google.com/group/sage-devel/browse_thread/thread/e207ce2206f0beee [https://groups.google.com/group/sage-devel/browse_thread/thread/e207ce2206f0beee] for a good counterargument, particularly the post from William Stein. The key in that case, I think, is that many of the authors come from a culture (the academic mathematics community) where crediting directly at the source is the norm and is highly valued. In such circumstances, it may be preferable to put author names into the source files, along with precise descriptions of what each author did, since the majority of potential contributors will expect that style of acknowledgement.

The important thing is to remember that there is a continuum between the core and the periphery of any project. The main source code files for the software are clearly part of the core, and should be considered as maintained by the community. On the other hand, companion tools or pieces of documentation may be the work of single individuals, who maintain them essentially alone, even though the works may be associated with, and even distributed with, the project. There is no need to apply a one-size-fits-all rule to every file, as long as the principle that community-maintained resources are not allowed to become individual territories is upheld.

The Automation Ratio

Try not to let humans do what machines could do instead. As a rule of thumb, automating a common task is worth at least ten times the effort a developer would spend doing that task manually one time. For very frequent or very complex tasks, that ratio could easily go up to twenty or even higher.

Thinking of yourself as a "project manager", rather than just another developer, may be a useful attitude here. Sometimes individual developers are too wrapped up in low-level work to see the big picture and realize that everyone is wasting a lot of effort performing automatable tasks manually. Even those who do realize it may not take the time to solve the problem: because each individual performance of the task does not feel like a huge burden, no one ever gets annoyed enough to do anything about it. What makes automation compelling is that the small burden is multiplied by the number of times each developer incurs it, and then *that* number is multiplied by the number of developers.

Here, I am using the term "automation" broadly, to mean not only repeated actions where one or two variables change each time, but any sort of technical infrastructure that assists humans. The minimum standard automation required to run a project these days was described in Chapter 3, *Technical Infrastructure*, but each project may have its own special problems too. For example, a group working on documentation might want to have a web site displaying the most up-to-date versions of the documents at all times. Since documentation is often written in a markup language like XML, there may be a compilation step, often quite intricate, involved in creating displayable or downloadable documents. Arranging a web site where such compilation happens automatically on every commit can be complicated and time-consuming—but it is worth it, even if it costs you a day or more to set up. The overall benefits of having up-to-date pages available at all times are huge, even though the cost of *not* having them might seem like only a small annoyance at any single moment, to any single developer.

Taking such steps eliminates not merely wasted time, but the griping and frustration that ensue when humans make missteps (as they inevitably will) in trying to perform complicated procedures manually. Multi-step, deterministic operations are exactly what computers were invented for; save your humans for more interesting things.

(Another example of using automation to remove a bottleneck for the entire team is Subversion's Contribulyzer system, which I've already described it in detail in Chapter 21 of the book Beautiful Teams [http://shop.oreilly.com/product/9780596518028.do]. That chapter, *Teams and Tools*, is available online at red-bean.com/kfogel/beautiful-teams/bt-chapter-21.html [http://red-bean.com/kfogel/beautiful-teams/bt-chapter-21.html].)

Automated testing

Automated test runs are helpful for any software project, but especially so for open source projects, because automated testing (especially regression testing) allows developers to feel comfortable changing code in areas they are unfamiliar with, and thus encourages exploratory development. Since detecting breakage is so hard to do by hand—one essentially has to guess where one might have broken something, and try various experiments to prove that one didn't—having automated ways to detect such breakage saves the project a *lot* of time. It also makes people much more relaxed about refactoring large swaths of code, and therefore contributes to the software's long-term maintainability.

Regression Testing

Regression testing means testing for the reappearance of already-fixed bugs. The purpose of regression testing is to reduce the chances that code changes will break the software in unexpected ways. As a software project gets bigger and more complicated, the chances of such unexpected side effects increase steadily. Good design can reduce the rate at which the chances increase, but it cannot eliminate the problem entirely.

As a result, many projects have a *test suite*, a separate program that invokes the project's software in ways that have been known in the past to stimulate specific bugs. If the test suite succeeds in making one of these bugs happen, this is known as a *regression*, meaning that someone's change unexpectedly unfixed a previously-fixed bug.

See also en.wikipedia.org/wiki/Regression_testing [https://en.wikipedia.org/wiki/Regression_testing].

Regression testing is not a panacea. For one thing, it works best for programs with batch-style interfaces. Software that is operated primarily through graphical user interfaces is much harder to drive programmatically. Another problem is that the regression test suite framework itself can often be quite complex, with a learning curve and maintenance burden all its own. Reducing this complexity is one of the most useful things you can do, even though it may take a considerable amount of time. The easier it is to add new tests to the suite, the more developers will do so, and the fewer bugs will survive to release. Any effort spent making tests easier to write will be paid back manyfold over the lifetime of the project.

Many projects have a *"Don't break the build!"* rule, meaning: don't commit a change that makes the software unable to compile or run. Being the person who broke the build is usually cause for mild embarrassment and ribbing. Projects with regression test suites often have a corollary rule: don't commit any change that causes tests to fail. Such failures are easiest to spot if there are automatic nightly runs of the entire test suite, with the results mailed out to the development list or to a dedicated test-results mailing list; that's another example of a worthwhile automation.

Most developers are willing to take the extra time to write regression tests, when the test system is comprehensible and easy to work with. Accompanying changes with tests is understood to be the responsible thing to do, and it's also an easy opportunity for collaboration: often two developers will divide up the work for a bugfix, with one writing the fix itself, and the other writing the test. The latter developer may often end up with more work, and since writing a test is already less satisfying than actually fixing the bug, it is imperative that the test suite not make the experience more painful than it has to be.

Some projects go even further, requiring that a new test accompany *every* bugfix or new feature. Whether this is a good idea or not depends on many factors: the nature of the software, the makeup of the development team, and the difficulty of writing new tests. The CVS (cvs.nongnu.org/ [http://cvs.nongnu.org/]) project has long had such a rule. It is a good policy in theory, since CVS is version control software and therefore very risk-averse about the possibility of munging or mishandling the user's data. The problem in practice is that CVS's regression test suite is a single huge shell script (amusingly named `sanity.sh`), hard to read and hard to modify or extend. The difficulty of adding new tests, combined with the requirement that patches be accompanied by new tests, means that CVS effectively discourages patches. When I used to work on CVS, I sometimes saw people start on and even complete a patch to CVS's own code, but give up when told of the requirement to add a new test to `sanity.sh`.

It is normal to spend more time writing a new regression test than on fixing the original bug. But CVS carried this phenomenon to an extreme: one might spend hours trying to design one's test properly, and still get it wrong, because there are just too many unpredictable complexities involved in changing a 35,000-line Bourne shell script. Even longtime CVS developers often grumbled when they had to add a new test.

This situation was due to a failure on all our parts to consider the automation ratio. It is true that switching to a real test framework—whether custom-built or off-the-shelf—would have been a major effort.[4] But neglecting to do so has cost the project much more, over the years. How many bugfixes and new features are *not* in CVS today, because of the impediment of an awkward test suite? We cannot know the exact number, but it is surely many times greater than the number of bugfixes or new features the developers might forgo in order to develop a new test system (or integrate an off-the-shelf system). That task would only take a finite amount of time, while the penalty of using the current test suite will continue forever if nothing is done.

The point is not that having strict requirements to write tests is bad, nor that writing your test system as a Bourne shell script is necessarily bad. It might work fine, depending on how you design it and what it needs to test. The point is simply that when the test system becomes a significant impediment to development, something must be done. The same is true for any routine process that turns into a barrier or a bottleneck.

Treat Every User as a Potential Participant

Each interaction with a user is an opportunity to get a new participant. When a user takes the time to post to one of the project's mailing lists, or to file a bug report, she has already tagged herself as having more potential for involvement than most users (from whom the project will never hear at all). Follow up on that potential: if she described a bug, thank her for the report and ask her if he wants to try fixing it. If she wrote to say that an important question was missing from the FAQ, or that the program's documentation was deficient in some way, then freely acknowledge the problem (assuming it really exists) and ask if she's interested in writing the missing material herself. Naturally, much of the time the user will demur. But it doesn't cost much to ask, and every time you do, it reminds the other listeners in that forum that getting involved in the project is something anyone can do.

Don't limit your goals to acquiring new developers and documentation writers. For example, even training people to write good bug reports pays off in the long run, if you don't spend *too* much time per person, and if they go on to submit more bug reports in the future—which they are more likely to do if they got a constructive reaction to their first report. A constructive reaction need not be a fix for the bug, although that's always the ideal; it can also be a solicitation for more information, or even just a confirmation that the behavior *is* a bug. People want to be listened to. Secondarily, they want their bugs fixed. You may not always be able to give them the latter in a timely fashion, but you (or rather, the project as a whole) can give them the former.

A corollary of this is that developers should not express anger at people who file well-intended but vague bug reports. This is one of my personal pet peeves; I see developers do it all the time on various open source mailing lists, and the harm it does is palpable. Some hapless newbie will post a useless report:

> Hi, I can't get Scanley to run. Every time I start it up, it just errors. Is anyone else seeing this problem?

Some developer—who has seen this kind of report a thousand times, and hasn't stopped to think that the newbie has not—will respond like this:

> What are we supposed to do with so little information? Sheesh. Give us at least some details, like the version of Scanley, your operating system, and the error.

This developer has failed to see things from the user's point of view, and also failed to consider the effect such a reaction might have on all the *other* people watching the exchange. Naturally a user who may have no programming experience, and no prior experience reporting bugs, will not know how to write a

[4]Note that there would be no need to convert all the existing tests to the new framework; the two could happily exist side by side, with old tests converted over only as they needed to be changed.

bug report. What is the right way to handle such a person? Educate them! And do it in such a way that they come back for more:

> Sorry you're having trouble. We'll need more information in order to figure out what's happening here. Please tell us the version of Scanley, your operating system, and the exact text of the error. The very best thing you can do is send a transcript showing the exact commands you ran, and the output they produced. See http://www.scanley.org/how_to_report_a_bug.html for more.

This way of responding is far more effective at extracting the needed information from the user, because it is written to the user's point of view. First, it expresses sympathy: *You had a problem; we feel your pain*. (This is not necessary in every bug report response; it depends on the severity of the problem and how upset the user seemed.) Second, instead of belittling him for not knowing how to report a bug, it tells him how, and in enough detail to be actually useful—for example, many users don't realize that "show us the error" means "show us the exact text of the error, with no omissions or abridgements." The first time you work with such a user, you need to be specific about that. Finally, it offers a pointer to much more detailed and complete instructions for reporting bugs. If you have successfully engaged with the user, he will often take the time to read that document and do what it says. This means, of course, that you have to have the document prepared in advance. It should give clear instructions about what kind of information your development team wants to see in every bug report. Ideally, it should also evolve over time in response to the particular sorts of omissions and misreports users tend to make for your project.

The Subversion project's bug reporting instructions, at subversion.apache.org/reporting-issues.html [http://subversion.apache.org/reporting-issues.html], are a fairly standard example of the form. Notice how they include an invitation to provide a patch to fix the bug. This is not because such an invitation will lead to a greater patch/report ratio—most users who are capable of fixing bugs already know that a patch would be welcome, and don't need to be told. The invitation's real purpose is to emphasize to all readers, especially those new to the project or new to free software in general, that the project runs on participation. In a sense, the project's current developers are no more responsible for fixing the bug than is the person who reported it. This is an important point that many new users will not be familiar with. Once they realize it, they're more likely to help make the fix happen, if not by contributing code then by providing a more thorough reproduction recipe, or by offering to test fixes that other people post. The goal is to make every user realize that there is no *innate* difference between himself and the people who work on the project—that it's a question of how much time and effort one puts in, not a question of who one is.

The admonition against responding angrily does not apply to rude users. Occasionally people post bug reports or complaints that, regardless of their informational content, show a sneering contempt at the project for some failing. Often such people are alternately insulting and flattering, such as the person who posted this to a Subversion mailing list:

> Why is it that after almost 6 days there still aren't any binaries posted for the windows platform?!? It's the same story every time and it's pretty frustrating. Why aren't these things automated so that they could be available immediately?? When you post an "RC" build, I think the idea is that you want users to test the build, but yet you don't provide any way of doing so. Why even have a soak period if you provide no means of testing??

Initial response to this rather inflammatory post was surprisingly restrained: people pointed out that the project had a published policy of not providing official binaries, and said, with varying degrees of annoyance, that he ought to volunteer to produce them himself if they were so important to him. Believe it or not, his next post started with these lines:

> First of all, let me say that I think Subversion is awesome and I really appreciate the efforts of everyone involved. [...]

...and then he went on to berate the project *again* for not providing binaries, while still not volunteering to do anything about it. After that, about 50 people just jumped all over him, and I can't say I really minded. Retaliatory rudeness should be avoided toward people with whom the project has (or would like to have) a sustained interaction. But when someone makes it clear from the start that he is going to be a fountain of bile, there is no point making him feel welcome.

Such situations are fortunately quite rare, and they are noticeably rarer in projects that make an effort to engage users constructively and courteously from their very first interaction.

Meeting In Person (Conferences, Hackfests, Code-a-Thons, Code Sprints, Retreats)

In the section called “Sponsoring Conferences, Hackathons, and other Developer Meetings” in Chapter 5, *Organizations, Money, and Business*, I already discussed the usefulness of sponsoring in-person meetings between developers, including those who are not part of your organization but who work on the same project(s) as your own developers do. Subsidizing in-person meetups, hackathons, and conference travel creates good will and is a relatively cheap way to signal the permanence of your company's strategic investestment in a given project.

Once you have decided to sponsor in-person contact, what form should it take?

The important thing to remember is that *the primary output of a social event is social connections*. Don't sponsor a hackathon with just the limited goal of getting a specific list of bugs fixed or features implemented. While it is reasonable to expect some technical progress as the result of a hackathon, if that's all you get, you're wasting at least some of your money. The *real* output is the increased trust and richer shared vocabulary built up between the developers from having been in the same room talking through the same problems — and from having relaxed over a good meal later that evening. That closer relationship will continue to pay off long after the event is over, in people's willingness to spend an extra hour reviewing a commit, evaluating a design proposal, or helping someone debug an unexpected problem. Deeper long-term collaboration is the goal; the event is just a means of getting there.

Meetups do not only have to be for writing code. Documentation sprints, user-testing and QA sprints, and primarily user-centric events such as install fests are all useful. However, be careful to distinguish clearly between purely developer-oriented events and events with a broader demographic, because the developers who attend will want to know what kind of mindset to be in. Designing, coding, and debugging require a specific kind of concentration and mental stance, and it helps developers a lot to know in advance whether the event they're going is expected to have an atmosphere conducive to that kind of concentration or not. *Both* kinds of events are useful for developers, and it's important for them to interact with and develop relationships with documenters, testers, users, sales engineers, etc. They just need to know what they're going to, so they can prepare accordingly.

Share Management Tasks as Well as Technical Tasks

Share the management burden as well as the technical burden of running the project. As a project becomes more complex, an increasing proportion of the work becomes about managing people and information flow. There is no reason not to share that burden, and sharing it does not necessarily require a top-down hierarchy either. In fact, what happens in practice tends to be more of a peer-to-peer network topology than a military-style command structure.

Sometimes management roles are formalized and sometimes they happen spontaneously. In the Subversion project, we have a patch manager, a translation manager, documentation managers, issue managers (albeit unofficial), and a release manager. Some of these roles we made a conscious decision to initiate,

others just happened by themselves. Here we'll examine these roles, and a couple of others, in detail (except for release manager, which was already covered in the section called “Release manager” and the section called “Dictatorship by Release Owner” earlier in this chapter).

"Manager" Does Not Mean "Owner"

As you read the role descriptions below, notice that none of them requires exclusive control over the domain in question. The issue manager does not prevent other people from making changes in the tickets database, the FAQ manager does not insist on being the only person to edit the FAQ, and so on. These roles are all about *responsibility without monopoly*. An important part of each domain manager's job is to notice when other people are working in that domain, and train them to do the things the way the manager does, so that the multiple efforts reinforce rather than conflict. Domain managers should also document the processes by which they do their work, so that when one leaves, someone else can pick up the slack right away.

Sometimes there is a conflict: two or more people want the same role. There is no one right way to handle this. You just have to draw on your knowledge of the project and of the people involved and suggest a resolution. In some cases it will work to just put on your "benevolent dictator" hat and choose one of the people. But I find that a better technique is just to ask the multiple candidates to settle it among themselves. They usually will, and will be more satisfied with the result than if a decision had been imposed on them from the outside. They may even decide on a co-management arrangement, which is fine if it works, and if it doesn't then you're right back where you started and can try a different resolution.

Patch Manager

In a free software project that receives a lot of patches, keeping track of which patches have arrived and what has been decided about them can be a nightmare, especially if done in a decentralized way. Most patches arrive either as posts to the project's development mailing list or as a pull request submitted through the version control system, but there are a number of different routes a patch can take after arrival.

Sometimes someone reviews the patch, finds problems, and bounces it back to the original author for cleanup. This usually leads to an iterative process—all visible in a public forum—in which the original author posts revised versions of the patch until the reviewer has nothing more to criticize. It is not always easy to tell when this process is done: if the reviewer commits the patch, then clearly the cycle is complete. But if she does not, it might be because she simply didn't have time, or doesn't have commit access herself and couldn't rope any of the other developers into doing it.

Another frequent response to a patch is a freewheeling discussion, not necessarily about the patch itself, but about whether the concept behind the patch is good. For example, the patch may fix a bug, but the project prefers to fix that bug in another way, as part of solving a more general class of problems. Often this is not known in advance, and it is the patch that stimulates the discovery.

Occasionally, a posted patch is met with utter silence. Usually this is due to no developer having time *at that moment* to review the patch, so each hopes that someone else will do it. Since there's no particular limit to how long each person waits for someone else to pick up the ball, and meanwhile other priorities are always coming up, it's very easy for a patch to be ignored permanently without any single person intending for that to happen. The project might miss out on a useful patch this way, and there are other harmful side effects as well: it is discouraging to the author, who invested work in the patch, and it is discouraging to others considering writing patches.

The patch manager's job is to make sure that patches don't "slip through the cracks." This is done by following every patch through to some sort of stable state. The patch manager watches every issue tracker discussion, pull request, or mailing list thread that results from a patch posting. If it ends with a commit of the patch, he does nothing. If it goes into a review/revise iteration, ending with a final version of

the patch but no commit, he creates or updates a ticket to point to the final version, and to any discussion around it, so that there is a permanent record for developers to follow up on later. In projects that use a patch queue manager or review tool (e.g., Gerrit [https://www.gerritcodereview.com/], Review_Board [https://en.wikipedia.org/wiki/Review_Board], etc), the patch manager can help encourage consistent usage of that tool, by putting patches there and watching to make sure developers handle them there.

When a patch gets no reaction at all, the patch manager waits a few days, then follows up asking if anyone is going to review it. This usually gets a reaction: a developer may explain that she doesn't think the patch should be applied, and give the reasons why, or she may review it, in which case one of the previously described paths is taken. If there is still no response, the patch manager may or may not file a ticket for the patch, at his discretion, but at least the original submitter got *some* reaction. The true currency of open source projects is attention: people who can see that they are getting attention will keep participating, even if not every patch they submit lands.

Having a patch manager has saved the Subversion development team a lot of time and mental energy. Without a designated person to take responsibility, every developer would constantly have to worry "If I don't have time to respond to this patch right now, can I count on someone else doing it? Should I try to keep an eye on it? But if other people are also keeping an eye on it, for the same reasons, then we'd have needlessly duplicated effort." The patch manager removes the second-guessing from the situation. Each developer can make the decision that is right for her at the moment she first sees the patch. If she wants to follow up with a review, she can do that—the patch manager will adjust his behavior accordingly. If she wants to ignore the patch completely, that's fine too; the patch manager will make sure it isn't forgotten.

Because this system works only if people can depend on the patch manager being there without fail, the role should be held formally. In Subversion, we advertised for it on the development and users mailing lists, got several volunteers, and took the first one who replied. When that person had to step down (see the section called “Transitions” later in this chapter), we did the same thing again. We've never tried having multiple people share the role, because of the communications overhead that would be required between them; but perhaps at very high volumes of patch submission, a multiheaded patch manager might make sense.

Translation Manager

In software projects, "translation" can refer to two somewhat different things. It can mean translating the software's documentation into other languages, or it can mean translating the software itself—that is, having the program display errors and help messages in the user's preferred language. Both are complex tasks, but once the right infrastructure is in place, they are largely separable from other development. Because the tasks are similar in some ways, it may make sense, depending on your project, to have a single translation manager handle both, or it may be better to have two different managers. (Note also that specialized infrastructure is available to help make the translation process more efficient; see the section called “Translation Infrastructure” in Chapter 3, *Technical Infrastructure* for more on this.)

In the Subversion project, we had one translation manager handle both. He did not actually write the translations himself, of course—he might help out on one or two, but would need to speak more than ten languages fluently in order to work on all of them! Instead, he managed teams of other translators: he helped them coordinate among each other, and he coordinated between the translation teams and the rest of the project.

Part of the reason the translation manager is necessary is that translators are a different demographic from developers. They sometimes have little or no experience working in a version control repository, or indeed with working as part of a distributed team at all. But in other respects they are often the best kind of participant: people with specific domain knowledge who saw a need and chose to get involved. They are usually willing to learn, and enthusiastic to get to work. All they need is someone to tell them how. The translation manager makes sure that the translations happen in a way that does not interfere unnec-

essarily with regular development. He also serves as a sort of representative of the translators as a unified body, whenever the developers must be informed of technical changes required to support the translation effort.

Thus, the position's most important skills are diplomatic, not technical. For example, in Subversion we had a policy that all translations should have at least two people working on them, because otherwise there is no way for the text to be reviewed. When a new person shows up offering to translate Subversion to, say, Malagasy, the translation manager has to either hook him up with someone who posted six months ago expressing interest in doing a Malagasy translation, or else politely ask the person to go find *another* Malagasy translator to work with as a partner. Once enough people are available, the manager sets them up with the proper kind of commit access, informs them of the project's conventions (such as how to write log messages), and then keeps an eye out to make sure they adhere to those conventions.

Conversations between the translation manager and the developers, or between the translation manager and translation teams, are usually held in the project's original language—that is, the language from which all the translations are being made. For most free software projects, this is English, but it doesn't matter what it is as long as the project agrees on it. (English is probably best for projects that want to attract a broad international development community, though.)

Conversations *within* a particular translation team usually happen in their shared language, however, and one of the other tasks of the translation manager is to set up a dedicated mailing list for each team. That way the translators can discuss their work freely, without distracting people on the project's main lists, most of whom would not be able to understand the translation language anyway.

Internationalization Versus Localization

Internationalization (*I18N*) and *localization* (*L10N*) both refer to the process of adapting a program to work in linguistic and cultural environments other than the one for which it was originally written. The terms are often treated as interchangeable, but in fact they are not quite the same thing. As https://en.wikipedia.org/wiki/Internationalization_and_localization writes:

> The distinction between them is subtle but important: Internationalization is the adaptation of products for *potential* use virtually everywhere, while localization is the addition of special features for use in a *specific* locale.

For example, changing your software to losslessly handle Unicode (en.wikipedia.org/wiki/Unicode [https://en.wikipedia.org/wiki/Unicode]) text encodings is an internationalization move, since it's not about a particular language, but rather about accepting text from any of a number of languages. On the other hand, making your software print all error messages in Slovenian, when it detects that it is running in a Slovenian environment, is a localization move.

Thus, the translation manager's task is principally about localization, not internationalization.

Documentation Manager

Keeping software documentation up-to-date is a never-ending task. Every new feature or enhancement that goes into the code has the potential to cause a change in the documentation. Also, once the project's documentation reaches a certain level of completeness, you will find that a lot of the patches people send in are for the documentation, not for the code. This is because there are many more people competent to fix bugs in prose than in code: all users are readers, but only a few are programmers.

Documentation patches are usually easier to review and apply than code patches. There is little or no testing to be done, and the quality of the change can be evaluated quickly just by examination. Since the quantity is high, but the review burden fairly low, the ratio of administrative overhead to productive work is greater for documentation patches than for code patches. Furthermore, most of the patches will

probably need some sort of adjustment, in order to maintain a consistent authorial voice in the documentation. In many cases, patches will overlap with or affect other patches, and need to be adjusted with respect to each other before being committed.

Given the exigencies of handling documentation patches, and the fact that the codebase needs to be constantly monitored so the documentation can be kept up-to-date, it makes sense to have one person, or a small team, dedicated to the task. They can keep a record of exactly where and how the documentation lags behind the software, and they can have practiced procedures for handling large quantities of patches in an integrated way.

Of course, this does not preclude other people in the project from applying documentation patches on the fly, especially small ones, as time permits. And the same patch manager (see the section called “Patch Manager” earlier in this chapter) can track both code and documentation patches, filing them wherever the development and documentation teams want them, respectively. (If the total quantity of patches ever exceeds one human's capacity to track, though, switching to separate patch managers for code and documentation is probably a good first step.) The point of a documentation team is to ensure that there are people who think of themselves as responsible for keeping the documentation organized, up-to-date, and consistent with itself. In practice, this means knowing the documentation intimately, watching the codebase, watching the changes *others* commit to the documentation, watching for incoming documentation patches, and using all these information sources to do whatever is necessary to keep the documentation healthy. If the documentation is kept in a wiki, then of course the wiki's "watch changes" feature can be very important to the documentation managers, since (depending on the wiki's edit policy) changes may land without going through a pre-change review process.

Issue Manager

Bug report growth is proportional to user base growth, rather than to the number of actual bugs in the software. That is, the number of tickets in a project's bug tracker grows in proportion to the number of people *using* the software.[5] Therefore, even as you fix bugs and ship an increasingly robust, mature program, you should still expect the number of open tickets to grow essentially without bound. The frequency of duplicate tickets will thus also increase, as will the frequency of incomplete or poorly described tickets.

An *issue manager*[6] helps cope with this situation by watching what goes into the database, and periodically sweeping through it looking for specific problems. Their most common action is probably to fix up incoming tickets, either because the reporter didn't set some of the form fields correctly, or because the ticket is a duplicate of one already in the database. Obviously, the more familiar an issue manager is with the project's bug database, and with the issue-tracking software's user interface and APIs, the more efficiently she will be able to detect and handle duplicate tickets. This is why it is often good to have a few people specialize in the bug database, instead of everyone trying to do it *ad hoc*. Although every developer in the project needs a certain basic level of competence in manipulating the issue tracker, having a few specialists becomes increasingly important as the project matures. When a project tries to spread collective responsibility for the bug database across everyone, no single individual acquires a deep enough expertise in the content of the database or the tracker's features.

Issue managers can help map between tickets and individual developers. When there are a lot of bug reports coming in, not every developer may read the ticket notification mailing list with equal attention. However, if someone who knows the development team is keeping an eye on all incoming tickets, then she can discreetly direct certain developers' attention to specific bugs when appropriate. Of course, this has to be done with a sensitivity to everything else going on in development, and to the recipient's desires and temperament. Therefore, it is often best for issue managers to be developers themselves.

[5] See rants.org/2010/01/10/bugs-users-and-tech-debt [http://www.rants.org/2010/01/10/bugs-users-and-tech-debt/] for a more detailed discussion of this.

[6] In the nomenclature I've been using elsewhere in this book, this position might be called "ticket manager", but in practice no project calls it that, and most call it "issue manager", so that's what we'll use here too.

Depending on how your project uses the ticket tracker, issue managers can also shape the database to reflect the project's priorities. For example, in Subversion we scheduled tickets into specific future releases, so that when someone asks "When will bug X be fixed?" we could say "Two releases from now," even if we can't give an exact date. The releases are represented in the ticket tracker as target milestones (something most ticket trackers support). As a rule, every Subversion release has one major new feature and a list of specific bug fixes. We assigned the appropriate target milestone to all the tickets planned for that release (including the new feature—it got a ticket too), so that people could view the bug database through the lens of release scheduling. These targets rarely remain static, however. As new bugs come in, priorities sometimes get shifted around, and tickets must be moved from one milestone to another so that each release remains manageable. This, again, is best done by people who have an overall sense of what's in the database, and how various tickets relate to each other.

Another thing issue managers do is notice when tickets become obsolete. Sometimes a bug is fixed accidentally as part of an unrelated change to the software, or sometimes the project changes its mind about whether a certain behavior is buggy. Finding obsoleted tickets is not easy: the only way to do it systematically is by making a sweep over all the tickets in the database. But full sweeps become less and less feasible over time, as the number of tickets grows. After a certain point, the only way to keep the database sane is to use a divide-and-conquer approach: categorize tickets immediately on arrival and direct them to the appropriate developer's or team's attention. The recipient then takes charge of the ticket for the rest of its lifetime, shepherding it to resolution or oblivion as necessary. When the database is that large, the issue manager becomes more of an overall coordinator, spending less time looking at each ticket herself and more time getting it into the right person's hands.

Transitions

From time to time, a person in a position of ongoing responsibility (e.g., patch manager, translation manager, etc.) will become unable to perform the duties of the position. It may be because the job turned out to be more work than he anticipated, or it may be due to other factors: a change in responsibilities at his job, a new baby, or whatever.

When a person gets swamped like this, he usually doesn't notice it right away. It happens by slow degrees, and there's no point at which he consciously realizes that he can no longer fulfill the duties of the role. Instead, the rest of the project just doesn't hear much from him for a while. Then there will suddenly be a flurry of activity, as he feels guilty for neglecting the project for so long and sets aside a night to catch up. Then you won't hear from him for a while longer, and then there might or might not be another flurry. But there's rarely an unsolicited formal resignation. To resign would mean openly acknowledging to himself that his circumstances have changed and that his ability to fulfill a committment has been permanently reduced. People are often reluctant to admit that.

Therefore, it's up to you and the others in the project to notice what's happening—or rather, not happening—and to ask the person what's going on. The inquiry should be friendly and 100% guilt-free. Your purpose is to find out a piece of information, not to make the person feel bad. Generally, the inquiry should be visible to the rest of the project, but if you know of some special reason why a private inquiry would be better, that's fine too. The main reason to do it publicly is so that if the person responds by saying that he won't be able to do the job anymore, there's a context established for your *next* public post: a request for a new person to fill that role.

Sometimes, a person is unable to do the job he's taken on, but is either unaware or unwilling to admit that fact. Of course, anyone may have trouble at first, especially if the responsibility is complex. However, if someone just isn't working out in the role he's taken on, even after everyone else has given all the help and suggestions they can, then the only solution is for him to step aside and let someone new have a try. And if the person doesn't see this himself, he'll need to be told. There's basically only one way to handle this, I think, but it's a multistep process and each step is important.

First, make sure you're not crazy. Privately talk to others in the project to see if they agree that the problem is as serious as you think it is. Even if you're already positive, this serves the purpose of letting others know that you're considering asking the person to step aside. Usually no one will object to that—they'll just be happy you're taking on the awkward task, so they don't have to!

Next, *privately* contact the person in question and tell him, kindly but directly, about the problems you see. Be specific, giving as many examples as possible. Make sure to point out how people had tried to help, but that the problems persisted without improving. You should expect this email to take a long time to write, but with this sort of message, if you don't back up what you're saying, you shouldn't say it at all. Say that you would like to find a someone new to fill the role, but also point out that there are many other ways to contribute to the project. At this stage, don't say that you've talked to others about it; nobody likes to be told that people were conspiring behind his back.

There are a few different ways things can go after that. The most likely reaction is that he'll agree with you, or at any rate not want to argue, and be willing to step down. In that case, suggest that he make the announcement himself, and then you can follow up with a post seeking a replacement.

Or, he may agree that there have been problems, but ask for a little more time (or for one more chance, in the case of discrete-task roles like release manager). How you react to that is a judgement call, but whatever you do, don't agree to it just because you feel like you can't refuse such a reasonable request. That would prolong the agony, not lessen it. There is often a very good reason to refuse the request, namely, that there have already been plenty of chances, and that's how things got to where they are now. Here's how I put it in a mail to someone who was filling the release manager role but was not really suited for it:

```
> If you wish to replace me with some one else, I will gracefully
> pass on the role to who comes next.  I have one request, which
> I hope is not unreasonable.  I would like to attempt one more
> release in an effort to prove myself.

I totally understand the desire (been there myself!), but in
this case, we shouldn't do the "one more try" thing.

This isn't the first or second release, it's the sixth or
seventh... And for all of those, I know you've been dissatisfied
with the results too (because we've talked about it before).  So
we've effectively already been down the one-more-try route.
Eventually, one of the tries has to be the last one... I think
[this past release] should be it.
```

In the worst case, the person may disagree outright. Then you have to accept that things are going to be awkward and plow ahead anyway. Now is the time to say that you talked to other people about it (but still don't say who until you have their permission, since those conversations were confidential), and that you don't think it's good for the project to continue as things are. Be insistent, but never threatening. Keep in mind that with most roles, the transition really happens the moment someone new starts doing the job, *not* the moment the old person stops doing it. For example, if the contention is over the role of, say, issue manager, at any point you and other influential people in the project can solicit for a new issue manager. It's not actually necessary that the person who was previously doing it stop doing it, as long as he does not sabotage (deliberately or otherwise) the efforts of the new person.

Which leads to a tempting thought: instead of asking the person to resign, why not just frame it as a matter of getting him some help? Why not just have two issue managers, or patch managers, or whatever the role is?

Although that may sound nice in theory, it is generally not a good idea. What makes the manager roles work—what makes them useful, in fact—is their centralization. Those things that can be done in a decentralized fashion are usually already being done that way. Having two people fill one managerial role introduces communications overhead between those two people, as well as the potential for slippery displacement of responsibility ("I thought you brought the first aid kit!" "Me? No, I thought *you* brought the first aid kit!"). Of course, there are exceptions. Sometimes two people work extremely well together, or the nature of the role is such that it can easily be spread across multiple people. But these are not likely to be applicable when you see someone flailing in a role he is not suited for. If he'd appreciated the problem in the first place, he would have sought such help before now. In any case, it would be disrespectful to let someone waste time continuing to do a job no one will pay attention to.

The most important factor in asking someone to step down is privacy: giving him the space to make a decision without feeling like others are watching and waiting. I once made the mistake—an obvious mistake, in retrospect—of mailing all three parties at once in order to ask Subversion's release manager to step aside in favor of two others who were ready to step up. I'd already talked to the two new people privately, and knew that they were willing to take on the responsibility. So I thought, naïvely and somewhat insensitively, that I'd save some time and hassle by sending one mail to all of them to initiate the transition. I assumed that the current release manager was already fully aware of the problems and would see the reasonableness of my point immediately.

I was wrong. The current release manager was very offended, and rightly so. It's one thing to be asked to hand off the job; it's another thing to be asked that *in front of* the people you'll hand it off to. Once I got it through my head why he was offended, I apologized. He eventually did step aside gracefully, and continues to be involved with the project today. But his feelings were hurt, and needless to say, this was not the most auspicious of beginnings for the new release managers either.

Committers

Defining "committer" and "commit access".

For the purposes of this section, the word *committer* means one of the official maintainers of the software — a committer is someone who has *commit access*: the right to make changes to the copy of the code that will be used for the project's next official release.

This precise definition is important because, after all, anyone can set up a repository containing a copy of the project's code and allow themselves to commit to that repository; indeed, doing so is a standard development procedure with decentralized version control systems such as Git. But what really matters for the project's purposes is who has the ability to put changes into the *master* copy — that is, the central shared copy into which contributors' changes are merged and from which releases are made.

Because in older, centralized version control systems, there was normally only one repository anyway, the term "commit access" corresponded closely to who was actually using the "commit" command (see *commit* in Chapter 3, *Technical Infrastructure*) to put changes into the group's shared repository. These days it corresponds to those who run the "push" or "pull" commands (see *push* and *pull* in Chapter 3, *Technical Infrastructure*) to put changes into that repository. It is the same idea either way: the master repository is a social concept, not a technical concept, and the mechanics of how changes get into it are not important here. Open source projects continue to use the term "committer" as a synonym for "official maintainer", even though formally speaking the "commit" command is no longer where the gating happens.

As the only formally distinct class of people found in all open source projects, committers deserve special attention in this book. Committers are an unavoidable concession to discrimination in a system which is otherwise as non-discriminatory as possible. But "discrimination" is not meant as a pejorative

here. The function committers perform is utterly necessary, and I do not think a project could succeed without it. Quality control requires, well, control. There are always many people who feel competent to make changes to a program, and some smaller number who actually are. The project cannot rely on people's own judgement; it must impose standards and grant commit access only to those who meet them. On the other hand, having people who can commit changes directly working side-by-side with people who cannot sets up an obvious power dynamic. That dynamic must be managed so that it does not harm the project.

In the section called “Who Votes?” in Chapter 4, *Social and Political Infrastructure*, we already discussed the mechanics of considering new committers. Here we will look at the standards by which potential new committers should be judged, and how this process should be presented to the larger community.

Choosing Committers

In the Subversion project, we choose committers primarily on the Hippocratic Principle: *first, do no harm*. Our main criterion is not technical skill or even knowledge of the code, but merely that the person show good judgement. Judgement includes knowing what not to take on. Someone might post only small patches, fixing fairly simple problems in the code, but if her patches apply cleanly, do not contain bugs, and are mostly in accord with the project's log message and coding conventions, and there are enough patches to show a clear pattern, then an existing committer will usually propose her for commit access. If at least three people say yes, and no one objects, then the offer is made. True, we might have no evidence that the person is able to solve complex problems in all areas of the codebase, but that is irrelevant: the person has made it clear that she is capable of judging her own abilities, and that is the important thing. Technical skills can be learned (and taught), but judgement, for the most part, cannot. Therefore, it is the one thing you want to make sure a person has before you give her commit access.

When a new committer proposal does provoke a discussion, it is usually not about technical ability, but rather about the person's behavior in the project's discussion forums. Sometimes someone shows technical skill and an ability to work within the project's formal guidelines, yet is also consistently belligerent or uncooperative in public forums. That's a serious concern; if the person doesn't seem to shape up over time, even in response to hints, then we won't add her as a committer no matter how skilled she is. In an open source project, social skills, or the ability to "play well in the sandbox", are as important as raw technical ability. Because everything is under version control, the penalty for adding a committer you shouldn't have added is not so much the problems it could cause in the code (review would spot those quickly anyway), but that it might eventually force the project to revoke the person's commit access—an action that is never pleasant and can sometimes be confrontational.

Some projects insist that the potential committer demonstrate a certain level of technical expertise and persistence, by submitting some number of nontrivial patches—that is, not only do these projects want to know that the person will do no harm, they want to know that she is likely to do good across the codebase. This isn't necessarily harmful, but be careful that it doesn't start to turn committership into a matter of membership in an exclusive club. The question to keep in everyone's mind should be "What will bring the best results for the code?" not "Will we devalue the social status associated with committership by admitting this person?" The point of commit access is not to reinforce people's self-worth, it's to allow good changes to enter the code with a minimum of fuss. If you have 100 committers, 10 of whom make large changes on a regular basis, and the other 90 of whom just fix typos and small bugs a few times a year, that's still better than having only the 10.

Revoking Commit Access

The first thing to be said about revoking commit access is: try not to be in that situation in the first place. Depending on whose access is being revoked, and why, the discussions around such an action can be very divisive. Even when not divisive, they will be a time-consuming distraction from productive work.

However, if you must do it, the discussion should be had privately among the same people who would be in a position to vote for *granting* that person whatever flavor of commit access they currently have. The person himself should not be included. This contradicts the usual injunction against secrecy, but in this case it's necessary. First, no one would be able to speak freely otherwise. Second, if the motion fails, you don't necessarily want the person to know it was ever considered, because that could open up questions ("Who was on my side? Who was against me?") that lead to the worst sort of factionalism. In certain rare circumstances, the group may want someone to know that revocation of commit access is or was being considered, as a warning, but this openness should be a decision the group makes. No one should ever, on her own initiative, reveal information from a discussion and ballot that others assumed were secret.

Once someone's access is revoked, that fact is unavoidably public (see the section called “Avoid Mystery” later in this chapter), so try to be as tactful as you can in how it is presented to the outside world.

Partial Commit Access

Some projects offer gradations of commit access. For example, there might be contributors whose commit access gives them free rein in the documentation, but who do not commit to the code itself. Common areas for partial commit access include documentation, translations, binding code to other programming languages, specification files for packaging (e.g., RedHat RPM spec files, etc.), and other places where a mistake will not result in a problem for the core project.

Since commit access is not only about committing, but about being part of an electorate (see the section called “Who Votes?” in Chapter 4, *Social and Political Infrastructure*), the question naturally arises: what can the partial committers vote on? There is no one right answer; it depends on what sorts of partial commit domains your project has. In Subversion things are fairly simple: a partial committer can vote on matters confined exclusively to that committer's domain, and not on anything else. Importantly, we do have a mechanism for casting advisory votes (essentially, the committer writes "+0" or "+1 (non-binding)" instead of just "+1" on the ballot). There's no reason to silence people just because their vote isn't formally binding.

Full committers can vote on anything, just as they can commit anywhere, and only full committers vote on adding new committers of any kind. In practice, though, the ability to add new partial committers is usually delegated: any full committer can "sponsor" a new partial committer, and partial committers in a domain can often essentially choose new committers for that same domain (this is especially helpful in making translation work run smoothly).

Your project may need a slightly different arrangement, depending on the nature of the work, but the same general principles apply to all projects. Each committer should be able to vote on matters that fall within the scope of her commit access, and not on matters outside that, and votes on procedural questions should default to the full committers, unless there's some reason (as decided by the full committers) to widen the electorate. Remember that voting should be rare anyway (see the section called “When To Vote” in Chapter 4, *Social and Political Infrastructure*), except for technical votes such as the change voting described in the section called “Voting on Changes” in Chapter 7, *Packaging, Releasing, and Daily Development*.

Regarding enforcement of partial commit access: it's often best *not* to have the version control system enforce partial commit domains, even if it is capable of doing so. See the section called “Authorization” in Chapter 3, *Technical Infrastructure* for the reasons why.

Dormant Committers

Some projects automatically remove people's commit access if they go a certain amount of time (say, a year) without committing anything. I think this is usually unhelpful and even counterproductive, for two reasons.

First, it may tempt some people into committing acceptable but unnecessary changes, just to prevent their commit access from expiring. Second, it doesn't really serve any purpose. If the main criterion for granting commit access is good judgement, then why assume someone's judgement would deteriorate just because she's been away from the project for a while? Even if she completely vanishes for years, not looking at the code or following development discussions, when she reappears she'll *know* how out of touch she is, and act accordingly. You trusted her judgement before, so why not trust it always? If high school diplomas do not expire, then commit access certainly shouldn't.

Sometimes a committer may ask to be removed, or to be explicitly marked as dormant in the list of committers (see the section called “Avoid Mystery” below for more about that list). In these cases, the project should accede to the person's wishes, of course.

Avoid Mystery

Although the discussions around adding any particular new committer must be confidential, the rules and procedures themselves need not be secret. In fact, it's best to publish them, so people realize that the committers are not some mysterious Star Chamber, closed off to mere mortals, but that anyone can join simply by posting good patches and knowing how to handle herself in the community. In the Subversion project, we put this information right in the developer guidelines document, since the people most likely to be interested in how commit access is granted are those thinking of contributing code to the project.

In addition to publishing the procedures, publish the actual *list* of committers. It often goes in a file called `MAINTAINERS` or `COMMITTERS` or something like that, in the top level of the project's source code tree. It should list all the full committers first, followed by the various partial commit domains and the members of each domain. Each person should be listed by name and email address, though the address can be encoded to prevent spam (see the section called “Address hiding in archives” in Chapter 3, *Technical Infrastructure*) if the person prefers that.

Since the distinction between full commit and partial commit access is obvious and well defined, it is proper for the list to make that distinction too. Beyond that, the list should not try to indicate the informal distinctions that inevitably arise in a project, such as who is particularly influential and how. It is a public record, not an acknowledgments file. List committers either in alphabetical order, or in the order in which they arrived.

Credit

Credit is the primary currency of the free software world. Whatever people may say about their motivations for participating in a project, I don't know many developers who would be happy doing all their work anonymously, or under someone else's name. There are tangible reasons for this: one's reputation in a project roughly governs how much influence one has, and participation in an open source project can also indirectly have monetary value, because many employers now look for it on resumés[7]. There are also intangible reasons, perhaps even more powerful: people simply want to be appreciated, and instinctively look for signs that their work was recognized by others. The promise of credit is therefore one of best motivators the project has. When small contributions are acknowledged, people come back to do more.

One of the most important features of collaborative development software (see Chapter 3, *Technical Infrastructure*) is that it keeps accurate records of who did what, when. Wherever possible, use these existing mechanisms to make sure that credit is distributed accurately, and be specific about the nature of the contribution. Don't just write "Thanks to J. Random <jrandom@example.com>" if instead you can

[7] Brian Fitzpatrick has written about the employment value of open source activity in *The Virtual Referral* (onlamp.com/pub/a/onlamp/2005/07/14/osdevelopers.html [http://www.onlamp.com/pub/a/onlamp/2005/07/14/osdevelopers.html]) and *The Virtual Internship* (onlamp.com/pub/a/onlamp/2005/08/01/opensourcedevelopers.html [http://www.onlamp.com/pub/a/onlamp/2005/08/01/opensourcedevelopers.html]).

write "Thanks to J. Random <jrandom@example.com> for the bug report and reproduction recipe" in a log message.

In Subversion, we have an informal but consistent policy of crediting the reporter of a bug in either the ticket filed, if there is one, or the log message of the commit that fixes the bug, if not. A quick survey of Subversion commit logs up to commit number 14525 shows that about 10% of commits give credit to someone by name and email address, usually the person who reported or analyzed the bug fixed by that commit. Note that this person is different from the developer who actually made the commit, whose name is already recorded automatically by the version control system. As of mid-2005, when I last did this calculation, slightly over # were credited in the commit logs, usually multiple times, before they became committers themselves. This does not, of course, prove that being credited was a factor in their continued involvement, but it surely can't hurt to set up an atmosphere in which people know they can count on their contributions being acknowledged. [8]

It is important to distinguish between routine acknowledgment and special thanks. When discussing a particular piece of code, or some other contribution someone made, it is fine to acknowledge their work. For example, saying "Daniel's recent changes to the delta code mean we can now implement feature X" simultaneously helps people identify which changes you're talking about and acknowledges Daniel's work. On the other hand, posting solely to thank Daniel for the delta code changes serves no immediate practical purpose. It doesn't add any information, since the version control system and other mechanisms have already recorded the fact that he made the changes. Thanking everyone for everything would be distracting and ultimately information-free, since thanks are effective largely by how much they stand out from the default, background level of favorable comment going on all the time. This does not mean, of course, that you should never thank people. Just make sure to do it in ways that tend not to lead to credit inflation. Following these guidelines will help:

- The more ephemeral the forum, the more free you should feel to express thanks there. For example, thanking someone for their bugfix in passing during an IRC conversation is fine, as is an aside in an email devoted mainly to other topics. But don't post a new email solely to thank someone, unless it's for a truly unusual feat, or if it's just one followup in a topic-specific thread already focused on the thing that person did.

  Likewise, don't clutter the project's web pages with expressions of gratitude. Once you start that, it'll never be clear when or where to stop. And *never* put thanks into comments in the code; that would only be a distraction from the primary purpose of comments, which is to help the reader understand the code.

- The less involved someone is in the project, the more appropriate it is to thank her for something she did. This may sound counterintuitive, but it fits with the attitude that expressing thanks is something you do when someone contributes even more than you thought she would. Thus, to constantly thank regular contributors for doing what they normally do would be to express a lower expectation of them than they have of themselves. If anything, you want to aim for the opposite effect!

  There are occasional exceptions to this rule. It's acceptable to thank someone for fulfilling his expected role when that role involves temporary, intense efforts from time to time. The canonical example is the release manager, who goes into high gear around the time of each release, but otherwise lies dormant (dormant as a release manager, in any case—he may also be an active developer, but that's a different matter).

[8] Eventually this crediting system became a bit more formalized, as described in subversion.apache.org/docs/community-guide/conventions.html#crediting [http://subversion.apache.org/docs/community-guide/conventions.html#crediting], thus improving the project's ability to find and encourage long-term participants, via a system known as the Contribulyzer. See the section called “The Automation Ratio” for more about this example.

- As with criticism and crediting, gratitude should be specific. Don't thank people just for being great, even if they are. Thank them for something they did that was out of the ordinary, and for bonus points, say exactly why what they did was so great.

In general, there is always a tension between making sure that people's individual contributions are recognized, and making sure the project is a group effort rather than a collection of individual glories. Just remain aware of this tension and try to err on the side of group, and things won't get out of hand.

Forks

"Social forks" versus "short forks".

At its most basic, a *fork* is when one copy of a project diverges from another copy: think "fork in the road".

What such divergence means for the project depends on the intentions behind the fork. There are two types of forks, which I will call *short forks* and *social forks*. The distinction between them is important.

Short forks are very common; in fact, they are the normal way development is done in most projects today. A developer creates her own public copy of the project's master repository, makes some changes, then submits the changes back to the project directly from the forked copy.[9] Short forks are done on a routine basis as part of daily development, and have no negative effect on the social cohesiveness of the project.

Social forks are much less common, and are much more significant when they happen. A social fork is when a group of developers is unhappy with the direction of the project and decides to create a divergent version more in line with their own vision. Of course, one of the technical actions required for such this is to create their own copy of the project's repository, and perhaps of its bug database and other resources as well. But this new copy of the project represents a potentially permanent divergence, and developers on both sides of the fork are aware of this; thus, it is a completely different beast from a cooperative short fork.

A social fork is almost always accompanied by long discussions and rationales, in which developers try to persuade each other of the merits of one or the other side of the fork, or of the merits of ending the fork and reunifying. Since social forks have implications for a project's stability and ability to continue attracting developers, knowing how to constructively initiate or react to a social fork of your project is useful — useful even if a fork never happens, as understanding what leads to social forks, and signalling clearly how you will behave in such an event, can sometimes prevent the fork from happening in the first place.

The rest of this section is about social forks, not short forks. To save space, I will just use the word "fork" instead of "social fork".

In the section called “Forkability” in Chapter 4, *Social and Political Infrastructure*, we saw how the *potential* to fork has important effects on how projects are governed. But what happens when a fork actually occurs? How should you handle it, and what effects can you expect it to have? Conversely, when should you *initiate* a fork?

The answers depend on what kind of fork it is. Some forks are due to amicable but irreconcilable disagreements about the direction of the project; perhaps more are due to both technical disagreements and interpersonal conflicts. Of course, it's not always possible to tell the difference between the two, as

[9] This is the "pull request [pull-requests]" workflow first popularized by GitHub.com. GitHub's decision to use the term "fork" instead of "clone" to refer to the personal copies in which development is done is largely responsible for the newer, "short fork" sense of "fork".

technical arguments may involve personal elements as well. What all forks have in common is that one group of developers (or sometimes even just one developer) has decided that the costs of working with some or all of the others now outweigh the benefits.

Once a project forks, there is no definitive answer to the question of which fork is the "true" or "original" project. People will colloquially talk of fork F coming out of project P, as though P is continuing unchanged down some natural path while F diverges into new territory, but this is, in effect, a declaration of how that particular observer feels about it. It is fundamentally a matter of perception: when a large enough percentage of observers agree, the assertion starts to become true. It is not the case that there is an objective truth from the outset, one that we are only imperfectly able to perceive at first. Rather, the perceptions *are* the objective truth, since ultimately a project—or a fork—is an entity that exists only in people's minds anyway.

If those initiating the fork feel that they are sprouting a new branch off the main project, the perception question is resolved immediately and easily. Everyone, both developers and users, will treat the fork as a new project, with a new name (perhaps based on the old name, but easily distinguishable from it), a separate web site, and a separate philosophy or goal. Things get messier, however, when both sides feel they are the legitimate guardians of the original project and therefore have the right to continue using the original name. If there is some organization with trademark rights to the name (see the section called “Trademarks” in Chapter 9, *Legal Matters: Licenses, Copyrights, Trademarks and Patents*), or legal control over the domain or web pages, that usually resolves the issue by fiat: that organization will decide who is the original project and who is the fork, because it holds all the cards in a public relations war. Naturally, things rarely get that far: since everyone already knows what the power dynamics are, they will avoid fighting a battle whose outcome is known in advance, and just jump straight to the end.

Fortunately, in most cases there is little doubt as to which is the project and which is the fork, because a fork is, in essence, a vote of confidence. If more than half of the developers are in favor of whatever course the fork proposes to take, usually there is no need to fork—the project can simply go that way itself, unless it is run as a dictatorship with a particularly stubborn dictator. On the other hand, if fewer than half of the developers are in favor, the fork is a clearly minority rebellion, and both courtesy and common sense indicate that it should think of itself as the divergent branch rather than the main line.

When a fork occurs, there can be a question of what happens to non-copyable assets: not just trademarks, but perhaps money in the bank, hardware, that full-color conference banner sitting in a storage locker somewhere, etc. Sometimes, those questions are resolved independently of the project's decision-making procdures, because those assets already had formal owners, and in each case the owner will decide what happens to the asset. But in cases where the actual ownership is in dispute, or the asset belongs in some way to the project as a whole, there is no magic answer. If someone decides to make a fuss, the dispute might wind up in a court of law. In this respect, open source projects are not different from any other endeavor involving multiple people: when agreement cannot be reached but no one is willing to give in, the last resort is the legal system. It is extremely rare, however, for things to go that far in a free software project (I can't think of any examples, actually), because usually there is no participant for whom going to court is a better option than just giving up their side of the argument anyway.

Handling a Fork

If someone threatens a fork in your project, keep calm and remember your long-term goals. The mere *existence* of a fork isn't what hurts a project; rather, it's the loss of developers and users. Your real aim, therefore, is not to squelch the fork, but to minimize these harmful effects. You may be mad, you may feel that the fork was unjust and uncalled for, but expressing that publicly can only alienate undecided developers. Instead, don't force people to make exclusive choices, and be as cooperative as is practicable with the fork.

Don't remove someone's commit access in your project just because she decided to work on the fork. Work on the fork doesn't mean that person has suddenly lost her competence to work on the origi-

nal project; committers before should remain committers afterward. Beyond that, you should express your desire to remain as compatible as possible with the fork, and say that you hope developers will port changes between the two whenever appropriate. If you have administrative access to the project's servers, publicly offer the forkers infrastructure help at startup time. For example, offer them a complete export of the bug database if there's no other way for them to get it. Ask them if there's anything else they need, and provide it if you can. Bend over backward to show that you are not standing in the way, and that you want the fork to succeed or fail on its own merits and nothing else.

The reason to do all this—and do it publicly—is not to actually help the fork, but to persuade developers that your side is a safe bet, by appearing as non-vindictive as possible. In war it sometimes makes sense (strategic sense, if not human sense) to force people to choose sides, but in free software it almost never does. In fact, after a fork some developers often openly work on both projects, doing their best to keep the two compatible. These developers help keep the lines of communication open after the fork. They allow your project to benefit from interesting new features in the fork (yes, the fork may have things you want), and also increase the chances of a merger down the road.

Sometimes a fork becomes so successful that, even though it was regarded even by its own instigators as a fork at the outset, it becomes the version everybody prefers, and eventually supplants the original by popular demand. A famous instance of this was the GCC/EGCS fork. The *GNU Compiler Collection* (*GCC*, formerly the *GNU C Compiler*) is the most popular open source native-code compiler, and also one of the most portable compilers in the world. Due to disagreements between the GCC's official maintainers and Cygnus Software,[10] one of GCC's most active developer groups, Cygnus created a fork of GCC called *EGCS*. The fork was deliberately non-adversarial: the EGCS developers did not, at any point, try to portray their version of GCC as a new official version. Instead, they concentrated on making EGCS as good as possible, incorporating patches at a faster rate than the official GCC maintainers. EGCS grew in popularity, and eventually some major operating system distributors decided to package EGCS as their default compiler instead of GCC. At this point, it became clear to the GCC maintainers that holding on to the "GCC" name while everyone switched to the EGCS fork would burden everyone with a needless name change, yet do nothing to prevent the switchover. So GCC adopted the EGCS codebase, and there is once again a single GCC, but greatly improved because of the fork.

This example shows why you cannot always regard a fork as an unadulteratedly bad thing. A fork may be painful and unwelcome at the time, but you cannot necessarily know whether it will succeed. Therefore, you and the rest of the project should keep an eye on it, and be prepared not only to absorb features and code where possible, but in the most extreme case to even join the fork if it gains the bulk of the project's mindshare. Of course, you will often be able to predict a fork's likelihood of success by seeing who joins it. If the fork is started by the project's biggest complainer and is joined by a handful of disgruntled developers who weren't behaving constructively anyway, they've essentially solved a problem for you by forking, and you probably don't need to worry about the fork taking momentum away from the original project. But if you see influential and respected developers supporting the fork, you should ask yourself why. Perhaps the project was being overly restrictive, and the best solution is to adopt into the mainline project some or all of the changes contemplated by the fork—in essence, to avoid the fork by becoming it.

Initiating a Fork

All the advice below assumes that you are forking as a last resort. Exhaust all other possibilities before starting a fork. Forking almost always means losing developers, with only an uncertain promise of gaining new ones later. It also means starting out with competition for users' attention: everyone who's about to install the software has to ask themselves: "Hmm, do I want that one or the other one?" Whichever one you are, the situation is messy, because a question has been introduced that wasn't there before. Some people maintain that forks are healthy for the software ecosystem as a whole, by a standard natural

[10] Now part of RedHat (redhat.com [https://www.redhat.com/]).

selection argument: the fittest will survive, which means that, in the end, everyone gets better software. This may be true from the ecosystem's point of view, but it's not true from the point of view of any individual project. Most forks do not succeed, and most projects are not happy to be forked.

A corollary is that you should not use the threat of a fork as an extremist debating technique—"Do things my way or I'll fork the project!"—because everyone is aware that a fork that fails to attract developers away from the original project is unlikely to survive long. All observers—not just developers, but users and operating system packagers too—will make their own judgement about which side to choose. You should therefore appear extremely reluctant to fork, so that if you finally do it, you can credibly claim it was the only route left.

Do not neglect to take *all* factors into account in evaluating the potential success of your fork. For example, if many of the developers on a project have the same employer, then even if they are disgruntled and privately in favor of a fork, they are unlikely to say so out loud if they know that their employer is against it. Many free software programmers like to think that having a free license on the code means no one company can dominate development. It is true that the license is, in an ultimate sense, a guarantor of freedom: if others want badly enough to fork the project, and have the resources to do so, they can. But in practice, some projects' development teams are mostly funded by one entity, and there is no point pretending that the entity's support doesn't matter. If it is opposed to the fork, its developers are unlikely to take part, even if they secretly want to.

If, after careful consideration, you still conclude that you must fork, line up support privately first, then announce the fork in a non-hostile tone. Even if you are angry at, or disappointed with, the current maintainers, don't say that in the message. Just dispassionately state what led you to the decision to fork, and that you mean no ill will toward the project from which you're forking. Assuming that you do consider it a fork (as opposed to an emergency preservation of the original project), emphasize that you're forking the code and not the name, and choose a name that does not conflict with the project's name. You can use a name that contains or refers to the original name, as long as it does not open the door to identity confusion. Of course it's fine to explain prominently on the fork's home page that it descends from the original program, and even that it hopes to supplant it. Just don't make users' lives harder by forcing them to untangle an identity dispute.

Finally, you can get things started on the right foot by automatically granting *all* committers of the original project commit access to the fork, including even those who openly disagreed with the need for a fork. Even if they never use the access, your message is clear: there are disagreements here, but no enemies, and you welcome code contributions from any competent source.

Chapter 9. Legal Matters: Licenses, Copyrights, Trademarks and Patents

Legal questions have assumed a somewhat more prominent role in free software projects over the last decade or so. It is still the case that the most important things about your project are its the quality of its code, its features, and the health of its developer community. However, although all open source licenses share the same basic guarantees of freedom, their terms are not exactly the same in all details. The particular license your project uses can affect which entities decide to get involved in it and how. You will therefore need a basic understanding of free software licensing, both to ensure that the project's license is compatible with its goals, and to be able to discuss licensing decisions with others.

Please note that I am not a lawyer, and that nothing in this book should be construed as formal legal advice. For that, you'll need to hire a lawyer or be one.[1]

Terminology

In any discussion of open source licensing, the first thing that becomes apparent is that there seem to be many different words for the same thing: *free software*, *open source*, *FOSS*, *F/OSS*, and *FLOSS*. Let's start by sorting those out, along with a few other terms.

free software

> Software that can be freely shared and modified, including in source code form. The term was first coined by Richard Stallman, who codified it in the GNU General Public License (GPL), and who founded the Free Software Foundation (fsf.org [https://www.fsf.org/]) to promote the concept.
>
> Although "free software" covers the same set of software as "open source", the FSF, among others, prefers the former term because it emphasizes the idea of freedom, and the concept of freely redistributable software as primarily a social movement rather than a technical one. The FSF acknowledges that the term is ambiguous—it could mean "free" as in "zero-cost", instead of "free" as in "freedom"—but feels that it's still the best term, all things considered, and that the other possibilities in English have their own ambiguities. (Throughout this book, "free" is used in the "freedom" sense, not the "zero-cost" sense.)

open source software

> Free software under another name. The different name is sometimes used to indicate a philosophical difference, however. In fact, the term "open source" was coined by the group that founded the Open Source Initiative (opensource.org [https://www.opensource.org/]) as a deliberate alternative to "free software". Their goal at the time was largely to make such software a more palatable choice for corporations, by presenting it as a development methodology rather than as a political movement.[2]
>
> While any license that is free is also open source, and vice versa (with a few minor exceptions that have no practical consequences), people tend to pick one term and stick with it. In general, those who prefer "free software" are more likely to have a philosophical or moral stance on the issue, while those who prefer "open source" either don't view it as a matter of freedom, or are not inter-

[1] For a deeper understanding of how copyright law relates to free software, see "Managing copyright information within a free software project" [https://softwarefreedom.org/resources/2012/ManagingCopyrightInformation.html], published by the Software Freedom Law Center.

[2] Disclaimer: Years after these events, I served as a member of the Board of Directors of the Open Source Initiative for three years, from 2011-2014. The ideological gap between the OSI and the FSF is much smaller these days than it was when the OSI was founded, in my opinion, and lately the two organizations have increasingly found common ground on which to cooperate. I remain a happy member of both, and urge you to join them too: opensource.org/join [https://opensource.org/join] and fsf.org/join [https://fsf.org/join].

ested in advertising the fact that they do. See the section called “"Free" Versus "Open Source"” in Chapter 1, *Introduction* for a more detailed history of this terminological schism.

The Free Software Foundation has an excellent—utterly unobjective, but nuanced and quite fair—exegesis of the two terms, at www.fsf.org/licensing/essays/free-software-for-freedom.html [https://www.fsf.org/licensing/essays/free-software-for-freedom.html]. The Open Source Initiative's take on it is (or was, in 2002) spread across two pages: web.archive.org/web/20021204155022/http://www.opensource.org/advocacy/case_for_hackers.php#marketing [https://web.archive.org/web/20021204155022/http://www.opensource.org/advocacy/case_for_hackers.php#marketing] and web.archive.org/web/20021204155022/http://www.opensource.org/advocacy/free-notfree.php [https://web.archive.org/web/20021204155022/http://www.opensource.org/advocacy/free-notfree.php] [sic].

FOSS, F/OSS, FLOSS

Where there are two of anything, there will soon be three, and that is exactly what is happening with terms for free software. The academic world, perhaps wanting precision and inclusiveness over elegance, seems to have settled on FOSS, or sometimes F/OSS, standing for "Free / Open Source Software". Another variant gaining momentum is FLOSS, which stands for "Free / Libre Open Source Software" (*libre* is familiar from many Romance languages and does not suffer from the ambiguities of "free"; see en.wikipedia.org/wiki/FLOSS [https://en.wikipedia.org/wiki/FLOSS] for more).

All these terms mean the same thing: software that can be modified and redistributed by everyone, sometimes—but not always—with the requirement that derivative works be freely redistributable under the same terms.

DFSG-compliant

Compliant with the Debian Free Software Guidelines (debian.org/social_contract#guidelines [https://www.debian.org/social_contract#guidelines]). This is a widely-used test for whether a given license is truly open source (free, *libre*, etc.). The Debian Project's mission is to maintain an entirely free operating system, such that someone installing it need never doubt that she has the right to modify and redistribute any or all of the system. The Debian Free Software Guidelines are the requirements that a software package's license must meet in order to be included in Debian. Because the Debian Project spent a good deal of time thinking about how to construct such a test, the guidelines they came up with have proven very robust (see en.wikipedia.org/wiki/DFSG [https://en.wikipedia.org/wiki/DFSG]), and as far as I'm aware, no serious objection to them has been raised either by the Free Software Foundation or the Open Source Initiative. If you know that a given license is DFSG-compliant, you know that it guarantees all the important freedoms (such as forkability even against the original author's wishes) required to sustain the dynamics of an open source project. Since 2004, the Debian Project has maintained a list of known DFSG-compliant licenses at wiki.debian.org/DFSGLicenses [https://wiki.debian.org/DFSGLicenses]. All of the licenses discussed in this chapter are DFSG-compliant.

OSI-approved

Approved by the Open Source Initiative. This is another widely-used test of whether a license permits all the necessary freedoms. The OSI's definition of open source software is based on the Debian Free Software Guidelines, and any license that meets one definition almost always meets the other. There have been a few exceptions over the years, but only involving niche licenses and none of any relevance here. The OSI maintains a list of all licenses it has ever approved, at opensource.org/licenses/ [https://www.opensource.org/licenses/], so that being "OSI-approved" is an unambiguous state: a license either is or isn't on the list.

The Free Software Foundation also maintains a list of licenses at fsf.org/licensing/licenses/license-list.html [https://www.fsf.org/licensing/licenses/license-list.html]. The FSF categorizes li-

censes not only by whether they are free, but whether they are compatible with the GNU General Public License. GPL compatibility is an important topic, covered in the section called “The GPL and License Compatibility” later in this chapter.

proprietary, closed-source

The opposite of "free" or "open source." It means software distributed under traditional, royalty-based licensing terms, where users pay per copy, or under any other terms sufficiently restrictive to prevent open source dynamics from operating. Even software distributed at no charge can still be proprietary, if its license does not permit free redistribution and modification.

Generally "proprietary" and "closed-source" are synonyms. However, "closed-source" additionally implies that the source code cannot even be seen. Since the source code cannot be seen with most proprietary software, this is normally a distinction without a difference. However, occasionally someone releases proprietary software under a license that allows others to view the source code. Confusingly, they sometimes call this "open source" or "nearly open source," etc., but that's misleading. The *visibility* of the source code is not the issue; the important question is what you're allowed to do with it: if you can't copy, modify, and redistribute, then it's not open source. Thus, the difference between proprietary and closed-source is mostly irrelevant; generally, the two can be treated as synonyms.

Sometimes *commercial* is used as a synonym for "proprietary," but this is carelessness: the two are not the same. Free software is always commercial software. After all, free software can be sold, as long as the buyers are not restricted from giving away copies themselves. It can be commercialized in other ways as well, for example by selling support, services, and certification. There are billion-dollar companies built on free software today, so it is clearly neither inherently anti-commercial nor anti-corporate. It is merely anti-proprietary, or if you prefer anti-monopoly, and this is the key way in which it differs from per-copy license models.

public domain

Having no copyright holder, meaning that there is no one who has the right to restrict copying of the work. Being in the public domain is not the same as having no author. Everything has an author, and even if a work's author or authors choose to put it in the public domain, that doesn't change the fact that they wrote it.

When a work is in the public domain, material from it can be incorporated into a copyrighted work, and the derivative is thus under the same overall copyright as the original copyrighted work. But this does not affect the availability of the original public domain wok. Thus, releasing something into the public domain is technically one way to make it "free," according to the guidelines of most free software certifying organizations (see opensource.org/faq#public-domain [https://opensource.org/faq#public-domain] for more). However, there are usually good reasons to use a license instead of just releasing into the public domain: even with free software, certain terms and conditions can be useful, not only to the copyright holder but to recipients as well, as the next section makes clear.

copyleft

A license that not only grants the freedoms under discussion here but furthermore requires that those freedoms apply to any derivative works.

The canonical example of a copyleft license is still the GNU General Public License, which stipulates that any derivative works must also be licensed under the GPL; see the section called “The GPL and License Compatibility” later in this chapter for more.

non-copyleft or *permissive*

A license that grants the freedoms under discussion here but that does *not* have a clause requiring that they apply to derivative works as well.

Two early and well-known examples of permissive licenses are the BSD and MIT licenses, but the more recent Apache Software License version 2 (apache.org/licenses/LICENSE-2.0 [https://www.apache.org/licenses/LICENSE-2.0]) is also very popular—increasingly so—and somewhat better adapted to the legal landscape of modern open source software development.

"Free Software" and "Open Source" Are the Same Licenses.

Occasionally people will make the mistake of thinking that copyleft licenses (like the GPL) comprise "free software", while the permissive licenses comprise "open source". This is wrong, but it comes up just often enough to be worth mentioning here. Both free software and open source include *both* the copyleft and non-copyleft licenses — this is something that all the license-certifying organizations, including the FSF, the OSI, and the Debian Project, have always agreed on. If you see someone, particularly a journalist, making this mistake, please politely correct them, perhaps by pointing them to this note (producingoss.com/en/terminology.html#free-open-same [http://producingoss.com/en/terminology.html#free-open-same]). The last thing we need is yet more terminological confusion in the free and open source software movement.

Aspects of Licenses

Although there are many different free software licenses available, in the important respects they all say the same things: that anyone can modify the code, that anyone can redistribute it both in original and modified form, and that the copyright holders and authors provide no warranties whatsoever (avoiding liability is especially important given that people might run modified versions without even knowing it). The differences between licences boil down to a few oft-recurring issues:

compatibility with proprietary licenses

The permissive (non-copyleft) free licenses allow the covered code to be used in proprietary programs. This does not affect the licensing terms of the proprietary program: it is still as proprietary as ever, it just happens to contain some code from a non-proprietary source. The Apache License, X Consortium License, BSD-style license, and the MIT-style license are all examples of proprietary-compatible licenses.

compatibility with other types of free licenses

Most of the commonly-used permissive free licenses are compatible with each other, meaning that code under one license can be combined with code under another, and the result distributed under either license without violating the terms of the other. Some of them are also compatible with some of the copyleft licenses, meaning that a work comprised of code under the permissive license and code under the copyleft license can be distributed as a combined work under the copyleft license (since that's the license that places more conditions), with the original code in each case remaining under its original license. Typically these compatibility issues come up between some permissive license and the GNU General Public License (or its variant the the section called “The GPL and License Compatibility”). This topic is discussed in more detail in the section called “The GPL and License Compatibility” later in this chapter.

enforcement of crediting

Some free licenses stipulate that any use of the covered code be accompanied by a notice, whose placement and display is usually specified, giving credit to the authors or copyright holders of the

code. These licenses are often still proprietary-compatible: they do not necessarily demand that the derivative work be free, merely that credit be given to the free code.

protection of trademark

A variant of credit enforcement. Trademark-protecting licenses specify that the name of the original software (or its copyright holders, or their institution, etc.) may *not* be used to identify derivative works, at least not without prior written permission. This restriction can be implemented purely via trademark law anyway, whether or not it is also stipulated by the copyright license, so such clauses can be somewhat legally redundant — in effect, they amplify a trademark infringement into a copyright infringement as well.

Although credit enforcement insists that a certain name be used, and trademark protection insists that it not be used, they are both expressions of the same concept: that the original code's reputation be preserved, and not tarnished by association.

patent snapback

Certain licenses (e.g., the GNU General Public License version 3, the Apache License version 2, the Mozilla Public License 2.0, and a few others) contain language designed to prevent people from using patent law to take away the rights granted under copyright law by the licenses. They require contributors to grant patent licenses along with their contribution, covering any patents licenseable by the contributor that would be infringed by their contribution (or by the incorporation of their contribution into the work as a whole). Then they go further: if someone using software under the license initiates patent litigation against another party, claiming that the covered work infringes, the initiator automatically *loses* all the patent grants otherwise provided for that work by the license, and in the case of the GPL-3.0 loses their right to distribute under the license altogether.

protection of "artistic integrity"

Some licenses (the Artistic License, used for the most popular implementation of the Perl programming language, and Donald Knuth's TeX license, for example) require that modification and redistribution be done in a manner that distinguishes clearly between the pristine original version of the code and any modifications. They permit essentially the same freedoms as other free licenses, but impose certain requirements that make the integrity of the original code easy to verify. These licenses have not caught on much beyond the specific programs they were made for, and will not be discussed in this chapter; they are mentioned here only for the sake of completeness. I do not recommend licensing new code under them.

Most of these stipulations are not mutually exclusive, and some licenses include several. The common thread among them is that they place demands on the recipient in exchange for the recipient's right to use and/or redistribute the code.

The GPL and License Compatibility

The sharpest dividing line in licensing is that between proprietary-incompatible and proprietary-compatible licenses, that is, between the copyleft licenses and everything else. The canonical example of a copyleft license is the GNU General Public License (along with its network-oriented variant, the Affero GNU General Public License or AGPL, introduced later in this chapter in the section called “The GNU Affero GPL: A Version of the GNU GPL for Server-Side Code”), and one of the most important considerations in choosing the GPL or AGPL is the extent to which it is compatible with other licenses. For brevity, I'll refer just to the GPL below, but most of this applies to the AGPL as well.

Because the primary goal of the GPL's authors is the promotion of free software, they deliberately crafted the license to make it impossible to mix GPLed code into proprietary programs. Specifically, among

the GPL's requirements (see fsf.org/licensing/licenses/gpl.html [https://www.fsf.org/licensing/licenses/gpl.html] for its full text) are these two:

1. Any derivative work—that is, any work containing a nontrivial amount of GPLed code—must itself be distributed under the GPL.

2. No additional restrictions may be placed on the redistribution of either the original work or a derivative work. (The exact language is: "You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License.")

With these conditions, the GPL succeeds in making freedom contagious. Once a program is copyrighted under the GPL, its terms of redistribution are *reciprocal*[3]—they are passed on to anything else the code gets incorporated into, making it effectively impossible to use GPLed code in closed-source programs. However, these same clauses also make the GPL incompatible with certain other free licenses. The usual way this happens is that the other license imposes a requirement—for example, a credit clause requiring the original authors to be mentioned in some way—that is incompatible with the GPL's "You may not impose any further restrictions..." language. From the point of view of the Free Software Foundation, these second-order consequences are desirable, or at least not regrettable. The GPL not only keeps your software free, but effectively makes your software an agent in pushing *other* software to enforce freedom as well.

The question of whether or not this is a good way to promote free software is one of the most persistent holy wars on the Internet (see the section called “Avoid Holy Wars” in Chapter 6, *Communications*), and we won't investigate it here. What's important for our purposes is that GPL compatibility is something to consider when choosing a license. The GPL is by far the most popular open source license, having more than twice as many projects released under it as under the next most popular licenses[4]. If you want your code to be able to be mixed freely with GPLed code—and there's a lot of GPLed code out there—then you should pick a GPL-compatible license. Most of the GPL-compatible open source licenses are also proprietary-compatible: that is, code under such a license can be used in a GPLed program, and it can be used in a proprietary program. Of course, the *results* of these mixings would not be compatible with each other, since one would be under the GPL and the other would be under a closed-source license. But that concern applies only to the derivative works, not to the code you distribute in the first place.

Fortunately, the Free Software Foundation maintains a list showing which licenses are compatible with the GPL and which are not, at gnu.org/licenses/license-list.html [https://www.gnu.org/licenses/license-list.html]. All of the licenses discussed in this chapter are present on that list, on one side or the other.

Choosing a License

When choosing a license to apply to your project, use an existing license instead of making up a new one. And don't just use any existing license — use one of the widely-used, well-recognized existing licenses.

Such licenses are familiar to many people already. If you use one of them, people won't feel they have to read the legalese in order to use your code, because they'll have already done so for that license a long time ago. Thus, you reduce or remove one possible barrier to entry for your project. They are also of a high quality: they are the products of much thought and experience; indeed most of them are revisions of previous versions of themselves, and the modern versions represent a great deal of accumulated legal and technical wisdom. Unless your project has truly unusual needs, it is unlikely you could do better, even with a team of lawyers at your disposal.

[3]Some people use the term *viral* to describe the GPL's contagiousness; they do not always mean this pejoratively, but I still prefer "reciprocal" because it's more descriptive and less connotative of disease.
[4]This statistic is based on an aggregation of several license count sources, combined with some reasonable definitional assumptions.

Below is a list of licenses that in my opinion meets these criteria; in parentheses are the standard formal abbreviation for the license and an authoritative URL for its full text. This list is not in order of preference, but rather in roughly descending order from strong copyleft at the top to completely non-copyleft at the bottom. The exact provisions of each license differ in various interesting ways (except for MIT and BSD, which differ only in uninteresting ways), and there isn't space here to explore all the possible ramifications of each for your project. However, many good discussions of that sort are available on the Internet; in particular the Wikipedia pages for these licenses generally give good overviews.

If you have nothing else to guide you and you want a copyleft license, then choose either the GPL-3.0 or the AGPL-3.0 — the difference between them will be discussed below — and if you want a non-copyleft license, choose the MIT license. I've put those licenses in boldface to reflect this:

- **GNU General Public License version 3** (`GPL-3.0`, gnu.org/licenses/gpl.html [https://www.gnu.org/licenses/gpl.html])
- **GNU Affero General Public License version 3** (`AGPL-3.0`, gnu.org/licenses/agpl.html [https://www.gnu.org/licenses/agpl.html])
- Mozilla Public License 2.0 (`MPL-2.0`, mozilla.org/MPL [https://www.mozilla.org/MPL/])
- GNU Library or "Lesser" General Public License version 3 (`LGPL-3.0`, gnu.org/licenses/lgpl.html [https://www.gnu.org/licenses/lgpl.html])
- Eclipse Public License 1.0 (`EPL-1.0`, eclipse.org/legal/epl-v10.html [https://www.eclipse.org/legal/epl-v10.html]) *(Note that version 2 of the EPL was almost ready as of mid-2014, and may be out by the time you read this.)*
- **MIT license** (`MIT`, opensource.org/licenses/MIT [https://opensource.org/licenses/MIT])
- Apache License 2.0 (`Apache-2.0`, apache.org/licenses/LICENSE-2.0 [https://apache.org/licenses/LICENSE-2.0])
- BSD 2-Clause ("Simplified" or "FreeBSD") license (`BSD-2-Clause`, opensource.org/licenses/BSD-2-Clause [https://opensource.org/licenses/BSD-2-Clause])

Note that there are some arguments for choosing the Apache License 2.0 as a default non-copyleft license, and they are nearly as compelling as those for choosing MIT. In the end, I come down in favor of MIT because it is extremely short, and both widely used and widely recognized[5]. While the Apache License 2.0 has the advantage of containing some explicit defenses against misuse of software patents, which might be important to your organization depending on the kind of project you're launching, the MIT license is fully compatible with all versions of the GNU General Public License, meaning that you can distributed, under any version of the GPL, mixed-provenance works that contain MIT-licensed code. The GPL-compatibility situation for the Apache License, on the other hand, is more complicated — by some interpretations, it is compatible with GPL version 3 only. Therefore, to avoid giving your downstream redistributors the headache of having to read sentences like the preceding ones, I just recommend the MIT license as the default non-copyleft license for anyone who doesn't have a reason to choose otherwise.

The mechanics of applying a license to your project are discussed in the section called “How to Apply a License to Your Software” in Chapter 2, *Getting Started*.

The GNU General Public License

If you prefer that your project's code not be used in proprietary programs, or if you at least don't care whether or not it can be used in proprietary programs, the GNU General Public License is a good choice.

[5] License choice trends for open source repositories on GitHub are a good source of information on this. See github.com/blog/1964-open-source-license-usage-on-github-com [https://github.com/blog/1964-open-source-license-usage-on-github-com] for statistics from March of 2015.

When writing a code library that is meant mainly to be used as part of other programs, consider carefully whether the restrictions imposed by the GPL are in line with your project's goals. In some cases—for example, when you're trying to unseat a competing, proprietary library that offers the same functionality—it may make more strategic sense to license your code in such a way that it can be mixed into proprietary programs, even though you would otherwise not wish this. The Free Software Foundation even fashioned an alternative to the GPL for such circumstances: the *GNU Lesser GPL*[6] The LGPL has looser restrictions than the GPL, and can be mixed more easily with non-free code. The FSF's page about the LGPL, gnu.org/licenses/lgpl.html [https://www.gnu.org/licenses/lgpl.html], has a good discussion of when to use it.

The "or any later version" Option: Future-Proofing the GPL.

The GPL has a well-known optional recommendation that you release software under the current version of the GPL while giving downstream recipients the option to redistribute it under any *later* (i.e., future) version. The way to offer this option is to put language like this in the license headers (see the section called “How to Apply a License to Your Software” in Chapter 2, *Getting Started*) of the actual source files:

> *This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License,* ***or (at your option) any later version.***

(Emphasis added.)

Whether you want to offer that option depends largely on how likely you think the Free Software Foundation is to make GPL revisions that you would approve of. I think the FSF has done a good job of that so far, and I generally do include that option when I use the GPL. That way I don't have to be responsible for updating the licenses myself forever — which is good, since I won't be around forever. Others can do it, either just to keep the software license up-to-date with legal developments, or to solve some future license compatibility problem that couldn't have been anticipated now (for example, see the compatibility discussion in the section called “The GNU Affero GPL: A Version of the GNU GPL for Server-Side Code” below).

Not everyone feels the same way, however; most notably, the Linux kernel is famously licensed under the GNU GPL version 2 *without* the "or any later version" clause, and influential kernel copyright holders, especially Linus Torvalds, have expressed clearly that they do not intend to move its license to version 3.0.

This book cannot answer the question of whether you should include the option or not. You now know that you have the choice, at least, and that different people come to different conclusions about it.

The GNU Affero GPL: A Version of the GNU GPL for Server-Side Code

In 2007, the Free Software Foundation released a variant of the GPL called the *GNU Affero GPL*[7]. Its purpose is to bring copyleft-style sharing provisions to the increasing amount of code being run as hosted services — that is, software that runs "in the cloud" on remote servers, that users interact with only over the network, and that therefore is never directly distributed to users as executable or source code.

[6] Originally named the *GNU Library GPL*, and later renamed by the FSF).

[7] The history of the license and its name is a bit complicated. The first version of the license was originally released by Affero, Inc, who based it on the GNU GPL version 2. At the time, this was commonly referred to as the AGPL. Later, the Free Software Foundation decided to adopt the idea, but by then they had released version 3 of their GNU GPL, so they based their new Affero-ized license on that and called it the "GNU AGPL". The old Affero license is now rarely used and is more or less deprecated, but to avoid ambiguity, say "AGPL-3.0" or "GNU AGPL" to make it clear that you're referring to the modern GNU version of the license.

Many such services use GPL'd software, often with extensive modifications, yet could avoid publishing their changes because they weren't actually distributing code.

The AGPL's solution to this was to take the GPL and add a "Remote Network Interaction" clause, stating "*...if you modify the Program, your modified version must prominently offer all users interacting with it remotely through a computer network ... an opportunity to receive the Corresponding Source of your version ... at no charge, through some standard or customary means of facilitating copying of software.*" This expanded the GPL's enforcement powers into the new world of application service providers. The Free Software Foundation recommends that the GNU AGPL 3.0 be used for any software that will commonly be run over a network.

Note that the AGPL-3.0 is not directly compatible with GPL-2.0, though it is compatible with GPL-3.0. Since most software licensed under GPL-2.0 includes the "or any later version" clause anyway, that software can just be shifted to GPL-3.0 if and when you need to mix it with AGPL-3.0 code. However, if you need to mix with programs licensed strictly under the GPL-2.0 (that is, programs licensed without the "or any later version" clause), the AGPL3.0 wouldn't be compatible with that.

Although the history of the AGPL-3.0 is a bit complicated, the license itself is simple: it's just the GPL-3.0 with one extra clause about network interaction. The Wikipedia article on the AGPL is excellent: en.wikipedia.org/wiki/Affero_General_Public_License [https://en.wikipedia.org/wiki/Affero_General_Public_License]

The Copyright Holder Is Special, Even In Copyleft Licenses

One common misunderstanding is that licensing your software under the GPL or AGPL requires you to provide source code to anyone who requests it under the terms of the license. But that's not quite how it works. If you are the *sole* copyright holder in a piece of software, then you are not bound by the copyright terms you chose, because (essentially) you can't be forced to sue yourself for copyright infringement. You can enforce the terms on others, but it's up to you to decide whether and when those terms apply to you. After all, because you had the software originally, you never "distributed" it to yourself and thus are not bound by the redistribution requirements of the license.

Of course, this only applies to situations where you own the whole copyright. If you include others' GPL- or AGPL-licensed code in your project and then distribute that project, you are not the sole copyright holder, and so you are as bound by the original terms as anyone else who uses and redistributes that code, either unmodified or as part of a derivative work.

Is the GPL free or not free?

One consequence of choosing the GPL (or AGPL) is the possibility—small, but not infinitely small—of finding yourself or your project embroiled in a dispute about whether or not the GPL is truly "free", given that it places some restrictions on how you redistribute the code—namely, the restriction that the code cannot be distributed under any other license. For some people, the existence of this restriction means the GPL is therefore "less free" than non-copyleft licenses. Where this argument usually goes, of course, is that since "more free" must be better than "less free" (after all, who's not in favor of freedom?), it follows that those licenses are better than the GPL.

This debate is another popular holy war (see the section called “Avoid Holy Wars” in Chapter 6, *Communications*). Avoid participating in it, at least in project forums. Don't attempt to prove that the GPL is less free, as free, or more free than other licenses. Instead, emphasize the specific reasons your project chose the GPL. If the recognizability of license was a reason, say that. If the enforcement of a free license on derivative works was also a reason, say that too, but refuse to be drawn into discussion about whether this makes the code more or less "free". Freedom is a complex topic, and there is little point talking about it if terminology is going to be used as a stalking horse for substance.

Since this is a book and not a mailing list thread, however, I will admit that I've never understood the "GPL is not free" argument. The only restriction the GPL imposes is that it prevents people from imposing *further* restrictions. To say that this results in less freedom has always seemed to me like saying that outlawing slavery reduces freedom, because it prevents some people from owning slaves.

(Oh, and if you do get drawn into a debate about it, don't raise the stakes by making inflammatory analogies.)

Contributor Agreements

There are three ways to handle copyright ownership for free code and documentation that were contributed to by many people. The first is to ignore the issue of copyright entirely (I don't recommend this). The second is to collect a *contributor license agreement* (*CLA*) from each person who works on the project, explicitly granting the project the right to use that person's contributions. This is usually enough for most projects, and the nice thing is that in some jurisdictions, CLAs can be sent in by email. The third way is to get actual *copyright assignment* (*CA* from contributors, so that the project (i.e., some legal entity, usually a nonprofit) is the copyright owner for everything. This way is the most burdensome for contributors, and some contributors simply refuse to do it; only a few projects still ask for assignment, and I don't recommend that any project require it these days.[8]

Note that even under centralized copyright ownership, the code[9] remains free, because open source licenses do not give the copyright holder the right to retroactively proprietize all copies of the code. So even if the project, as a legal entity, were to suddenly turn around and start distributing all the code under a restrictive license, that wouldn't necessarily cause a problem for the public community. The other developers can start a fork based on the latest free copy of the code, and continue as if nothing had happened.

Doing Nothing

Some projects never collect CLAs or CAs from their contributors. Instead, they accept code whenever it seems reasonably clear that the contributor intended it to be incorporated into the project.

This can seem to work for a long time, as long as the project has no enemies. But I don't recommend it. Someone may eventually decide to sue for copyright infringement, alleging that they are the true owner of the code in question and that they never agreed to its being distributed by the project under an open source license. For example, the SCO Group did something like this to the Linux project, see en.wikipedia.org/wiki/SCO-Linux_controversies [https://en.wikipedia.org/wiki/SCO-Linux_controversies] for details. When this happens, the project will have no documentation showing that the contributor formally granted the right to use the code, which could make some legal defenses more difficult.

Contributor License Agreements

CLAs probably offer the best tradeoff between safety and convenience. A CLA is typically an electronic form that a developer fills out and sends in to the project, or even a web-based checkbox that the developer checks before completing their first contribution to the project. In many jurisdictions, such email submission or an online form is enough, though you should consult with a lawyer to see what method would be best for your project.

Some projects use two slightly different CLAs, one for individuals, and one for corporate contributors. But in both types, the core language is the same: the contributor grants the project a *"...perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare de-*

[8] Also, actual copyright transferral is subject to national law, and licenses designed for the United States may encounter problems elsewhere (e.g., in Germany, where it's apparently not possible to transfer copyright).
[9] I'll use "code" to refer to both code and documentation, from now on.

rivative works of, publicly display, publicly perform, sublicense, and distribute [the] Contributions and such derivative works." Again, you should have a lawyer approve any CLA, but if you get all those adjectives into it, you're off to a good start.

When you request CLAs from contributors, make sure to emphasize that you are *not* asking for actual copyright assignment. In fact, many CLAs start out by reminding the reader of this, for example like so:

> *This is a license agreement only; it does not transfer copyright ownership and does not change your rights to use your own Contributions for any other purpose.*

Developer Certificates of Origin (DCO): A Simpler Style of CLA

More and more projects are now using a particularly convenient style of simple CLA known as a *Developer Certificate of Origin* (*DCO*).

A DCO is essentially an attestation that the contributor intends to contribute the enclosed code under the project's license, and that the contributor has the right to do so. The contributor indicates her understanding of the DCO once, early on, for example by emailing its text from her usual contribution address to a special archive at the project[10]. Thereafter, the contributor includes a "Signed-Off-By:" line in her patches or commits, using the same identity, to indicate that the corresponding contribution is certified under the DCO. This gives the project the legal cover it needs, while giving contributors a low-bureaucracy process for submitting their contributions. The DCO relies on the project's native open source license for any trademark or patent provisions, which in most cases is fine.

The simplification that makes DCOs work so well is that they set the inbound license of the contribution to be the same as the outbound license of the project. This avoids the sticky issues that a more complex CLA can create, whereby the recipient of the CLA might reserve the right to relicense the project (and thus all the past contributions) under some different license in the future, possibly even a proprietary license. DCOs are perfect for a free software project whose contributors expect it to stay that way.

Proprietary Relicensing

Some companies offer open source code with a *proprietary relicensing* scheme[11], in which an open source version of the software is available under the usual open source terms, while a proprietary version is available for a fee.

Why would anyone want a proprietary version, when an open source version is already out there? There are two separate answers, reflecting the two different types of proprietary relicensing.

The first kind is about *selling exceptions* to copyleft requirements, and is typically used with code libraries rather than with standalone applications. The way it works is that the library's owner (i.e., copyright holder), seeing that some of the library's users want to incorporate it into their own proprietary applications, sells them a promise to *not* enforce the redistribution requirements of the open source version's license. This only works when the open source code is under a copyleft-style license, of course — in practice it is usually the GPL or AGPL.

With this promise in hand, the downstream users can use the library in their proprietary product without worry that they might be forced to share the source code to their full product under the copyleft license. One well-known example of "selling exceptions" is the MySQL database engine, which is distributed under the GPL version 2, but with a proprietary license offering available for many years, first from the Swedish company MySQL AB, and later from Oracle, Inc, which purchased MySQL AB in 2008.

[10] The DCO text is provided by the project, but you don't have to write your own from scratch; see developercertificate.org [http://developercertificate.org/] for example.

[11] This is sometimes also called *dual licensing*, but that term is ambiguous, as it has historically also referred to releasing open source software under two or more open source licenses simultaneously. I am grateful to Bradley Kuhn for pointing out this ambiguity and suggesting the more accurate term.

The second kind of proprietary relicensing, sometimes called the *freemium* or *open core* model, uses an open source version to drive sales of a presumably fancier proprietary version (see the section called “"Commercial" vs "Proprietary"” in Chapter 5, *Organizations, Money, and Business* for a discussion of some marketing pitfalls to avoid in this situation). Usually the company offering the proprietary version is also the primary maintainer of the open source version, in the sense of supplying most of the developer attention (this is usually inevitable, for reasons we'll get to in a moment). Furthermore, although in theory the company *could* offer paid support for both the open source and proprietary versions[12], in practice they almost always offer it only for the proprietary version, because then they can charge two fees: a subscription fee for the software itself and a fee for the support services, with only the latter having any marginal cost to the supplier.

You might be wondering: how can the copyright holder offer the software under a proprietary license if the terms of the GNU GPL stipulate that the code must be available under less restrictive terms? The answer is that the GPL's terms are something the copyright holder imposes on everyone else; the owner is therefore free to decide *not* to apply those terms to itself. In other words, one always has the right to not sue one's self for copyright infringement. This right is not tied to the GPL or any other open source license; it is simply in the nature of copyright law.

Problems with Proprietary Relicensing

Proprietary relicensing, of both varieties, tends to suffer from several problems.

First, it discourages the normal dynamics of open source projects, because any code contributors from outside the company are now effectively contributing to two distinct entities: the free version of the code and the proprietary version. While the contributor will be comfortable helping the free version, since that's the norm in open source projects, she may feel less enthusiastic about her contributions being useable in a monopolized proprietary product. That is, unlike a straight non-copyleft license by which anyone has the right to use the code as part of a proprietary work, here only *one* party has that right, and other participants in the project are thus being asked to contribute to an asymmetric result. This awkwardness is reflected and in some ways amplified by the fact that in a proprietary relicensing scheme, the copyright owner must collect some kind of formal agreement from each contributor (see the section called “Contributor Agreements” earlier in this chapter), in order to have the right to redistribute that contributor's code under a proprietary license. Because such an agreement needs to give the collecting entity special, one-sided rights that a typical open source contributor agreement doesn't include, the process of collecting agreements starkly confronts contributors with the imbalance of the situation, and some of them may decline to sign. (Remember, they don't need to sign a contribution agreement in order to distribute their own changes along with the original code; rather, the *company* needs the agreement in order to redistribute the contributor's changes, especially under a proprietary license. Asymmetry cuts both ways.)

Historically, many companies that have started out offering a seemingly clear proprietary relicensing option — use our product under open source terms, or buy a proprietary license so you can use it under proprietary terms — eventually graduated to something closer to a "shakedown" model instead, in which anyone who makes commercially significant use of the code ends up being pressured to purchase a proprietary license as a way of protecting their commercial revenue stream from harassment. The precise legal bases on which this pressure rests differ from case to case, but the overall pattern of behavior has been remarkably consistent.

Naturally, neither the companies initiating these shakedowns nor the parties who are its targets, most of whom eventually capitulate, have anything to gain from going on the record about it, so I can only tell you that I have heard of it informally and off-the-record from multiple sources, at different projects and different companies. One reason I generally advise companies who are serious about open source devel-

[12] In both cases hosted as Software-as-a-Service (SaaS), just to be clear.

opment to stay away from proprietary relicensing is that, if history is a reliable guide, the temptation to undermine the open source license will be overwhelming to the point of being impossible to resist.

Finally, there is a deep motivational problem for open source projects that operate in the shadow of a proprietarily relicensed version: the sense that most of the salaried development attention is going to the proprietary version anyway, and that therefore spending time contributing to the open source version is a fool's game — that one is just helping a commercial entity free up its own developers to work on features that the open source community will never see. This fear is reasonable on its face, but it also becomes a self-fulfilling prophecy: as more outside developers stay away, the company sees less reason to invest in the open source codebase, because they're not getting a community multiplier effect anyway. Their disengagement in turn discourages outside developers, and so on.

What seems to happen in practice is that companies that offer proprietarily relicensed software do not get truly active development communities with external participants. They get occasional small-scale bug fixes and cleanup patches from the outside, but end up doing most of the hard work with internal resources. Since this book is about running free software projects, I will just say that in my experience, proprietary relicensing schemes inevitably have a negative effect on the level of community engagement and the level of technical quality on the open source side. If you conclude that for business reasons you want to try it anyway, then I hope this section will at least help you mitigate some of those effects.[13]

Trademarks

Trademark law as applied to open source projects does not differ significantly from trademark law as applied elsewhere. This sometimes surprises people: they think that if the code can be copied freely, then that can't possibly be consistent with some entity controlling a trademark on the project's name or logo. It is consistent, however, and below I'll explain why, furnishing some examples.

First, understand what trademarks are about: they are about truth in labeling and, to some degree, endorsement. A trademarked name or symbol is a way for an entity — the entity who owns or controls that trademark — to signal, in an easily recognizeable way, that they approve of a particular product. Often they are signaling their approval because they are the source of the product, and purchases of that product provide a revenue stream for them. But that is not the only circumstance under which someone might want to enforce accurate attribution. For example, certification marks are trademarked names or symbols that an entity applies to *someone else's* product, in order to signal that the product meets the certifying entity's standards.

Importantly, *trademarks do not restrict copying, modification, or redistribution*. I cannot emphasize this enough: trademark is unrelated to copyright, and does not govern the same actions that copyright governs. Trademark is about what you may publicly call things, not about what you may do with those things nor with whom you may share them.

One famous example of trademark enforcement in free and open source software demonstrates these distinctions clearly.

[13]Sometimes the terms-of-service agreements for online software distribution services — the Apple App Store, for example — effectively force you to use proprietary relicensing if you want to distribute copylefted software. I won't go into detail here, but if you're distributing GPL-licensed or other copylefted code from a place that restricts users from redistributing what they download, you may be in this situation. For more information, see Steven J. Vaughan-Nichols' article "No GPL Apps for Apple's App Store" [http://www.zdnet.com/article/no-gpl-apps-for-apples-app-store/], Richard Gaywood's followup article "The GPL, the App Store, and you" [http://www.engadget.com/2011/01/09/the-gpl-the-app-store-and-you/], and Pieter Colpaert's explanation of how the iRail and BeTrains projects used *pro forma* dual-licensing to get around the problem, "About Apple store, GPL's, VLC and BeTrains" [https://bonsansnom.wordpress.com/2011/01/08/about-apple-store-gpls-vlc-and-betrains/]. Thanks to reader Nathan Toone for pointing out this problem.

Case study: Mozilla Firefox, the Debian Project, and Iceweasel

The Mozilla Foundation owns the trademarked name "Firefox", which it uses to refer to its popular free software web browser of the same name. The Debian Project, which maintains a long-running and also quite popular GNU/Linux distribution, wanted to package Firefox for users of Debian GNU/Linux.

So far, so good: Debian does not need Mozilla's permission to package Firefox, since Firefox is open source software. However, Debian does need Mozilla's permission to *call* the packaged browser "Firefox" and to use the widely-recognized Firefox logo (you've probably seen it: a long reddish fox curling its body and tail around a blue globe) as the icon for the program, because those are trademarks owned by Mozilla.

Normally, Mozilla would have happily given its permission. After all, having Firefox distributed in Debian is good for Mozilla's mission of promoting openness on the Web. However, various technical and policy effects of the Debian packaging process left Debian unable to fully comply with Mozilla's trademark usage requirements, and as a result, Mozilla informed Debian that their Firefox package could not use the Firefox name or branding. No doubt Mozilla did so with some reluctance, as it is not ideal for them to have their software used without clear attribution. However, they could have given Debian a trademark license and yet chose not to; presumably, this is because Debian was doing something with the code that Mozilla did not want accruing to their own reputation.[14]

This decision by Mozilla did not mean that Debian had to remove Firefox from their package system, of course. Debian simply changed the name to "Iceweasel" and used a different logo. The underlying code is still the Mozilla Firefox code, except for the minor bits Debian had to change to integrate the different name and logo — changes they were perfectly free to make, of course, because of the code's open source license.

It is even consistent to license your project's logo artwork files under a fully free license while still retaining a trademark on the logo, as the following story of the GNOME logo and the fish pedicure shop (I'm not making this up) illustrates.

Case study: The GNOME Logo and the Fish Pedicure Shop

The GNOME Project [https://gnome.org/], which produces one of the major free software desktop environments, is represented legally by the GNOME Foundation [https://www.gnome.org/foundation/], which owns and enforces trademarks on behalf of the project. Their best-known trademark is the GNOME logo: a curved, stylized foot with four toes floating close above it.[15]

One day, Karen Sandler, then the Executive Director of the GNOME Foundation, heard from a GNOME contributor that a mobile fish-pedicure van (fish pedicure is a technique in which one places one's feet in water so that small fish can nibble away dead skin) was using a modified version of the GNOME logo. The central foot part of the image had been slightly modified to look like a fish, and a fifth toe had been added above, so that the overall logo looked even more like a human foot but cleverly made reference to fish as well. You can see it, along with discussion of other trademark issues GNOME has dealt with, in

[14] In fact, that was indeed the reason, though we do not need to go into the details here of exactly what changes Debian makes to the Firefox code that Mozilla disagrees with strongly enough to want to dissociate their name from the result. The entire saga is recounted in more detail at en.wikipedia.org/wiki/Mozilla_Corporation_software_rebranded_by_the_Debian_project [https://en.wikipedia.org/wiki/Mozilla_Corporation_software_rebranded_by_the_Debian_project]. Coincidentally, I'm writing these words on a Debian GNU/Linux system, where Iceweasel has long been my default browser — I just used it to check that URL.

[15] You can see examples at gnome.org/foundation/legal-and-trademarks [https://www.gnome.org/foundation/legal-and-trademarks/].

the Linux Weekly News article where this story is told in full: lwn.net/Articles/491639 [https://lwn.net/Articles/491639/].

Although GNOME does actively enforce its trademarks, Sandler did not see any infringement in this case: the fish-pedicure business is so distant from what the GNOME Project does that there was no possibility of confusion in the mind of the public or dilution (if you'll pardon the expression) of the mark. Furthermore, because the *copyright* license on GNOME's images is an open source license, the fish pedicure company was free to make their modifications to the graphic and display the results. There was no trademark violation, because there was no infringement within GNOME's domain of activity, and there was no copyright violation, because GNOME's materials are released under free licenses.

The point of these examples is to merely show that there is no inherent contradiction in registering and maintaining trademarks related to open source projects. This does not mean that a trademark owner should do whatever they want with the marks, ignoring what other participants in the project have to say. Trademarks are like any other centrally-controlled resource: if you use them in a way that harms a significant portion of the project's community, then expect complaints and pushback in return; if you use them in a way that supports the goals of the project, then most participants will be glad and will consider that use to be itself a form of contribution.

Patents

Software patents have long been a lightning rod issue in free software, because they pose the only real threat against which the free software community cannot defend itself. Copyright and trademark problems can always be gotten around. If part of your code looks like it may infringe on someone else's copyright, you can just rewrite that part while continuing to use the same underlying algorithm. If it turns out someone has a trademark on your project's name, at the very worst you can just rename the project. Although changing names would be a temporary inconvenience, it wouldn't matter in the long run, since the code itself would still do what it always did.

But a patent is a blanket injunction against implementing a certain idea. It doesn't matter who writes the code, nor even what programming language is used. Once someone has accused a free software project of infringing a patent, the project must either stop implementing that particular feature, or expose the project *and its users* to expensive and time-consuming lawsuits. Since the instigators of such lawsuits are usually corporations with deep pockets—that's who has the resources and inclination to acquire patents in the first place—most free software projects cannot afford either to defend themselves nor to indemnify their users, and must capitulate immediately even if they think it highly likely that the patent would be unenforceable in court. To avoid getting into such a situation in the first place, free software projects have sometimes had to code defensively, avoiding patented algorithms in advance even when they are the best or only available solution to a programming problem.

Surveys and anecdotal evidence show that not only the vast majority of open source programmers, but a majority of *all* programmers, think that software patents should be abolished entirely.[16] Open source programmers tend to feel particularly strongly about it, and may refuse to work on projects that are too closely associated with the collection or enforcement of software patents. If your organization collects software patents, then make it clear, in a public and irrevocable way, that the patents would never be enforced when the infringement comes from open source code, and that the patents are only to be used as a defense in case some other party initiates an infringement suit against your organization. This is not only the right thing to do, it's also good open source public relations.[17]

Unfortunately, collecting patents purely for defensive purposes is rational. The current patent system, at least in the United States, is by its nature an arms race: if your competitors have acquired a lot of

[16] See groups.csail.mit.edu/mac/projects/lpf/Whatsnew/survey.html [https://groups.csail.mit.edu/mac/projects/lpf/Whatsnew/survey.html] for one such survey.

[17] For example, RedHat has pledged that open source projects are safe from its patents, see redhat.com/legal/patent_policy.html [https://www.redhat.com/legal/patent_policy.html].

patents, then your best defense is to acquire a lot of patents yourself, so that if you're ever hit with a patent infringement suit you can respond with a similar threat—then the two parties usually sit down and work out a cross-licensing deal so that neither of them has to pay anything, except to their patent lawyers of course.

The harm done to free software by software patents is more insidious than just direct threats to code development, however. Software patents encourage an atmosphere of secrecy among firmware designers, who justifiably worry that by publishing details of their interfaces they will be making it easier for competitors to find ways to slap them with patent infringement suits. This is not just a theoretical danger; it has apparently been happening for a long time in the video card industry, for example. Many video card manufacturers are reluctant to release the detailed programming specifications needed to produce high-performance open source drivers for their cards, thus making it impossible for free operating systems to support those cards to their full potential. Why would the manufacturers withold these specs? It doesn't make sense for them to work *against* software support; after all, compatibility with more operating systems can only mean more card sales. But it turns out that, behind the design room door, these shops are all violating one another's patents, sometimes knowingly and sometimes accidentally. The patents are so unpredictable and so potentially broad that no card manufacturer can ever be certain it's safe, even after doing a patent search. Thus, manufacturers dare not publish their full interface specifications, since that would make it much easier for competitors to figure out whether any patents are being infringed. (Of course, the nature of this situation is such that you will not find a written admission from a primary source that it is going on; I learned it through a personal communication.)

Modern free software licenses generally have clauses to combat, or at least mitigate, the dangers arising from software patents. Usually these clauses work by automatically revoking the overall open source license for any party who makes a patent infringement claim based on either the work as a whole[18], or based on the claimant's code contributions to the project. But though it is useful, both legally and politically, to build patent defenses into free software licenses in this way, in the end these protections are not be enough to dispel the chilling effect that the threat of patent lawsuits has on free software. Only changes in the substance or interpretation of international patent law will do that.

Recent developments, such as the 2014 decision by the U.S. Supreme Court against the patentability of abstract ideas, in *Alice Corp. v. CLS Bank* (en.wikipedia.org/wiki/Alice_Corp._v._CLS_Bank_International [https://en.wikipedia.org/wiki/Alice_Corp._v._CLS_Bank_International]), have made the future of software patents unpredictable. But there is so much money to be extracted via infringement claims, in particular by patent trolls [https://en.wikipedia.org/wiki/Patent_troll] but in general by any entity with a large patent portfolio and a lack of other revenue sources, that I am not optimistic this fight will be over any time soon. If you want to learn more about the problem, there are good links in the Wikipedia article en.wikipedia.org/wiki/Software_patent [https://en.wikipedia.org/wiki/Software_patent]. I've also written some blog posts summarizing the arguments against software patents, collected at www.rants.org/patent-posts [http://www.rants.org/patent-posts/]. As of this writing it's been about six years since the main posts there were published, but all the reasons why software patents are a bad idea are just as true now as they were then.

Further Resources

This chapter has only been an introduction to free software licensing, trademark, and patent issues. Although I hope it contains enough information to get you started on your own open source project, any serious investigation of legal issues will quickly exhaust what this book can provide. Here are some other resources:

- opensource.org/licenses [https://opensource.org/licenses]

[18]Remember that a patent may cover, or "read on" in patent jargon, code that the patent owner did not themselves write. It is thus not necessary for a party to have contributed code to an open source in order to claim patent infringement *by* that project.

The OSI license introduction page is a well-maintained source of information about widely used open source licenses, and offers answers to frequently asked questions. It's a good place to start if you have a general idea of what open source licenses do, but now need more information, for example to choose a license for your project.

- *Intellectual Property and Open Source: A Practical Guide to Protecting Code* by Van Lindberg. Published by O'Reilly Media, first edition July 2008, ISBN: 978-0-596-51796-0

  This is a full-length book on open source licensing, trademarks, patents, contracting, and more. It goes into much deeper detail than I could in this chapter. shop.oreilly.com/product/9780596517960.do [http://shop.oreilly.com/product/9780596517960.do] for details.

- *Make Your Open Source Software GPL-Compatible. Or Else.* by Dr. David A. Wheeler, at dwheeler.com/essays/gpl-compatible.html [http://www.dwheeler.com/essays/gpl-compatible.html].

  This is a detailed and well-written article on why it is important to use a GPL-compatible license even if you don't use the GPL itself. The article also touches on many other licensing questions, and has a high density of excellent links.

Appendix A. Canned Hosting Sites

In earlier editions of this book, before 2015, I maintained a list of credible, well-established canned hosting sites. I've stopped maintaining a complete list, as it's too easy for it to become out of date. The most important such sites — which are also the ones likely to be around for a long time — are discussed in the section called “Canned Hosting” anyway, and a more complete list is available at en.wikipedia.org/wiki/Comparison_of_open_source_software_hosting_facilities [https://en.wikipedia.org/wiki/Comparison_of_open_source_software_hosting_facilities].

Appendix B. Obsolete Appendix (was: Free Version Control Systems)

I used to maintain here a list of all the free software version control systems I knew anything about, with comments about some of them. While there is still experimentation in open source version control systems, at least for now we appear to have settled on a few standards: Git [https://git-scm.com], primarily, and somewhat more rarely Mercurial [https://www.mercurial-scm.org/] and sometimes Subversion [http://subversion.apache.org/]. My thoughts on which to choose for your project are already given in the section called “Choosing a Version Control System”, and it's not sustainable for me to maintain an increasingly out-of-date list of other version control systems that you are unlikely to use anyway. If you want a list, though, https://en.wikipedia.org/wiki/List_of_revision_control_software is a good place to start.

Appendix C. Obsolete Appendix (was: Free Bug Trackers)

I used to maintain here a list of all the open source bug trackers I knew anything about, with comments on some of them. Bug trackers are still an active area of development and experimentation, perhaps even more so than version control systems (see Appendix B, *Obsolete Appendix (was: Free Version Control Systems)*), and I've found it increasingly unsustainable to maintain an appendix that stays up-to-date with the field. While I have personal preferences — I particularly like Redmine [http://www.redmine.org/] — the important thing is that whatever tracker you choose, it have the features needed to support an active open source community; those features are discussed in detail in the section called “Bug Tracker”.

For lists of bug trackers and comparisons between them, try Wikipedia: Comparison of Issue Tracking Systems [https://en.wikipedia.org/wiki/Comparison_of_issue-tracking_systems].

Appendix D. Obsolete Appendix: (was: Why Should I Care What Color the Bikeshed Is?)

This appendix used to contain the complete text of Poul-Henning Kamp's famous "bikeshed" post, an excerpt from which appears in Chapter 6, *Communications*). It is an eloquent disquisition on what tends to go wrong in group discussions, but to save space I no longer include it as an appendix in this book. You can view the entire post at bikeshed.org [http://www.bikeshed.org/]. You can also point people to bikeshed.com [http://bikeshed.com]. Or blue.bikeshed.com [http://blue.bikeshed.com/], or yellow.bikeshed.com [http://yellow.bikeshed.com/], or teal.bikeshed.com [http://teal.bikeshed.com/], or mauve.bikeshed.com [http://mauve.bikeshed.com/], or tope.bikeshed.com [http://tope.bikeshed.com/], or...

Appendix E. Obsolete Appendix (was: Example Instructions for Reporting Bugs)

This appendix used to contain a lightly-edited copy of the Subversion project's online instructions to new users on how to report bugs. They're still good instructions, and are referenced from the section called “Treat Every User as a Potential Participant” in Chapter 8, *Managing Participants*. But as they are a living document, I felt it didn't make sense to include a frozen snaphot of them here, and so no longer maintain a copy in this appendix. You can view them at subversion.apache.org/reporting-issues.html [http://subversion.apache.org/reporting-issues.html].

Appendix F. Copyright

fullest extent possible.

Using Creative Commons Public Licenses

Creative Commons public licenses provide a standard set of terms and conditions that creators and other rights holders may use to share original works of authorship and other material subject to copyright and certain other rights specified in the public license below. The following considerations are for informational purposes only, are not exhaustive, and do not form part of our licenses.

> Considerations for licensors: Our public licenses are intended for use by those authorized to give the public permission to use material in ways otherwise restricted by copyright and certain other rights. Our licenses are irrevocable. Licensors should read and understand the terms and conditions of the license they choose before applying it. Licensors should also secure all rights necessary before applying our licenses so that the public can reuse the material as expected. Licensors should clearly mark any material not subject to the license. This includes other CC-licensed material, or material used under an exception or limitation to copyright. More considerations for licensors:
> wiki.creativecommons.org/Considerations_for_licensors

> Considerations for the public: By using one of our public licenses, a licensor grants the public permission to use the licensed material under specified terms and conditions. If the licensor's permission is not necessary for any reason--for example, because of any applicable exception or limitation to copyright--then that use is not regulated by the license. Our licenses grant only permissions under copyright and certain other rights that a licensor has authority to grant. Use of the licensed material may still be restricted for other reasons, including because others have copyright or other rights in the material. A licensor may make special requests, such as asking that all changes be marked or described. Although not required by our licenses, you are encouraged to respect those requests where reasonable. More_considerations for the public:
> wiki.creativecommons.org/Considerations_for_licensees

==========

Creative Commons Attribution-ShareAlike 4.0 International Public License

By exercising the Licensed Rights (defined below), You accept and agree to be bound by the terms and conditions of this Creative Commons Attribution-ShareAlike 4.0 International Public License ("Public License"). To the extent this Public License may be interpreted as a contract, You are granted the Licensed Rights in consideration of Your acceptance of these terms and conditions, and the Licensor grants You such rights in consideration of benefits the Licensor receives from

making the Licensed Material available under these terms and conditions.

Section 1 -- Definitions.

a. Adapted Material means material subject to Copyright and Similar Rights that is derived from or based upon the Licensed Material and in which the Licensed Material is translated, altered, arranged, transformed, or otherwise modified in a manner requiring permission under the Copyright and Similar Rights held by the Licensor. For purposes of this Public License, where the Licensed Material is a musical work, performance, or sound recording, Adapted Material is always produced where the Licensed Material is synched in timed relation with a moving image.

b. Adapter's License means the license You apply to Your Copyright and Similar Rights in Your contributions to Adapted Material in accordance with the terms and conditions of this Public License.

c. BY-SA Compatible License means a license listed at creativecommons.org/compatiblelicenses, approved by Creative Commons as essentially the equivalent of this Public License.

d. Copyright and Similar Rights means copyright and/or similar rights closely related to copyright including, without limitation, performance, broadcast, sound recording, and Sui Generis Database Rights, without regard to how the rights are labeled or categorized. For purposes of this Public License, the rights specified in Section 2(b)(1)-(2) are not Copyright and Similar Rights.

e. Effective Technological Measures means those measures that, in the absence of proper authority, may not be circumvented under laws fulfilling obligations under Article 11 of the WIPO Copyright Treaty adopted on December 20, 1996, and/or similar international agreements.

f. Exceptions and Limitations means fair use, fair dealing, and/or any other exception or limitation to Copyright and Similar Rights that applies to Your use of the Licensed Material.

g. License Elements means the license attributes listed in the name of a Creative Commons Public License. The License Elements of this Public License are Attribution and ShareAlike.

h. Licensed Material means the artistic or literary work, database, or other material to which the Licensor applied this Public License.

i. Licensed Rights means the rights granted to You subject to the terms and conditions of this Public License, which are limited to all Copyright and Similar Rights that apply to Your use of the Licensed Material and that the Licensor has authority to license.

j. Licensor means the individual(s) or entity(ies) granting rights under this Public License.

k. Share means to provide material to the public by any means or process that requires permission under the Licensed Rights, such as reproduction, public display, public performance, distribution, dissemination, communication, or importation, and to make material available to the public including in ways that members of the public may access the material from a place and at a time individually chosen by them.

l. Sui Generis Database Rights means rights other than copyright resulting from Directive 96/9/EC of the European Parliament and of the Council of 11 March 1996 on the legal protection of databases, as amended and/or succeeded, as well as other essentially equivalent rights anywhere in the world.

m. You means the individual or entity exercising the Licensed Rights under this Public License. Your has a corresponding meaning.

Section 2 -- Scope.

a. License grant.

1. Subject to the terms and conditions of this Public License, the Licensor hereby grants You a worldwide, royalty-free, non-sublicensable, non-exclusive, irrevocable license to exercise the Licensed Rights in the Licensed Material to:

a. reproduce and Share the Licensed Material, in whole or in part; and

b. produce, reproduce, and Share Adapted Material.

2. Exceptions and Limitations. For the avoidance of doubt, where Exceptions and Limitations apply to Your use, this Public License does not apply, and You do not need to comply with its terms and conditions.

3. Term. The term of this Public License is specified in Section 6(a).

4. Media and formats; technical modifications allowed. The Licensor authorizes You to exercise the Licensed Rights in all media and formats whether now known or hereafter created, and to make technical modifications necessary to do so. The Licensor waives and/or agrees not to assert any right or authority to forbid You from making technical modifications necessary to exercise the Licensed Rights, including technical modifications necessary to circumvent Effective Technological Measures. For purposes of this Public License, simply making modifications authorized by this Section 2(a)

(4) never produces Adapted Material.

5. Downstream recipients.

   a. Offer from the Licensor -- Licensed Material. Every recipient of the Licensed Material automatically receives an offer from the Licensor to exercise the Licensed Rights under the terms and conditions of this Public License.

   b. Additional offer from the Licensor -- Adapted Material. Every recipient of Adapted Material from You automatically receives an offer from the Licensor to exercise the Licensed Rights in the Adapted Material under the conditions of the Adapter's License You apply.

   c. No downstream restrictions. You may not offer or impose any additional or different terms or conditions on, or apply any Effective Technological Measures to, the Licensed Material if doing so restricts exercise of the Licensed Rights by any recipient of the Licensed Material.

6. No endorsement. Nothing in this Public License constitutes or may be construed as permission to assert or imply that You are, or that Your use of the Licensed Material is, connected with, or sponsored, endorsed, or granted official status by, the Licensor or others designated to receive attribution as provided in Section 3(a)(1)(A)(i).

b. Other rights.

1. Moral rights, such as the right of integrity, are not licensed under this Public License, nor are publicity, privacy, and/or other similar personality rights; however, to the extent possible, the Licensor waives and/or agrees not to assert any such rights held by the Licensor to the limited extent necessary to allow You to exercise the Licensed Rights, but not otherwise.

2. Patent and trademark rights are not licensed under this Public License.

3. To the extent possible, the Licensor waives any right to collect royalties from You for the exercise of the Licensed Rights, whether directly or through a collecting society under any voluntary or waivable statutory or compulsory licensing scheme. In all other cases the Licensor expressly reserves any right to collect such royalties.

Section 3 -- License Conditions.

Your exercise of the Licensed Rights is expressly made subject to the

following conditions.

a. Attribution.

1. If You Share the Licensed Material (including in modified form), You must:

a. retain the following if it is supplied by the Licensor with the Licensed Material:

i. identification of the creator(s) of the Licensed Material and any others designated to receive attribution, in any reasonable manner requested by the Licensor (including by pseudonym if designated);

ii. a copyright notice;

iii. a notice that refers to this Public License;

iv. a notice that refers to the disclaimer of warranties;

v. a URI or hyperlink to the Licensed Material to the extent reasonably practicable;

b. indicate if You modified the Licensed Material and retain an indication of any previous modifications; and

c. indicate the Licensed Material is licensed under this Public License, and include the text of, or the URI or hyperlink to, this Public License.

2. You may satisfy the conditions in Section 3(a)(1) in any reasonable manner based on the medium, means, and context in which You Share the Licensed Material. For example, it may be reasonable to satisfy the conditions by providing a URI or hyperlink to a resource that includes the required information.

3. If requested by the Licensor, You must remove any of the information required by Section 3(a)(1)(A) to the extent reasonably practicable.

b. ShareAlike.

In addition to the conditions in Section 3(a), if You Share Adapted Material You produce, the following conditions also apply.

1. The Adapter's License You apply must be a Creative Commons license with the same License Elements, this version or later, or a BY-SA Compatible License.

2. You must include the text of, or the URI or hyperlink to, the

Adapter's License You apply. You may satisfy this condition in any reasonable manner based on the medium, means, and context in which You Share Adapted Material.

3. You may not offer or impose any additional or different terms or conditions on, or apply any Effective Technological Measures to, Adapted Material that restrict exercise of the rights granted under the Adapter's License You apply.

Section 4 -- Sui Generis Database Rights.

Where the Licensed Rights include Sui Generis Database Rights that apply to Your use of the Licensed Material:

a. for the avoidance of doubt, Section 2(a)(1) grants You the right to extract, reuse, reproduce, and Share all or a substantial portion of the contents of the database;

b. if You include all or a substantial portion of the database contents in a database in which You have Sui Generis Database Rights, then the database in which You have Sui Generis Database Rights (but not its individual contents) is Adapted Material,

including for purposes of Section 3(b); and

c. You must comply with the conditions in Section 3(a) if You Share all or a substantial portion of the contents of the database.

For the avoidance of doubt, this Section 4 supplements and does not replace Your obligations under this Public License where the Licensed Rights include other Copyright and Similar Rights.

Section 5 -- Disclaimer of Warranties and Limitation of Liability.

a. UNLESS OTHERWISE SEPARATELY UNDERTAKEN BY THE LICENSOR, TO THE EXTENT POSSIBLE, THE LICENSOR OFFERS THE LICENSED MATERIAL AS-IS AND AS-AVAILABLE, AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE LICENSED MATERIAL, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHER. THIS INCLUDES, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OR ABSENCE OF ERRORS, WHETHER OR NOT KNOWN OR DISCOVERABLE. WHERE DISCLAIMERS OF WARRANTIES ARE NOT ALLOWED IN FULL OR IN PART, THIS DISCLAIMER MAY NOT APPLY TO YOU.

b. TO THE EXTENT POSSIBLE, IN NO EVENT WILL THE LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY (INCLUDING, WITHOUT LIMITATION, NEGLIGENCE) OR OTHERWISE FOR ANY DIRECT, SPECIAL, INDIRECT, INCIDENTAL, CONSEQUENTIAL, PUNITIVE, EXEMPLARY, OR OTHER LOSSES, COSTS, EXPENSES, OR DAMAGES ARISING OUT OF THIS PUBLIC LICENSE OR USE OF THE LICENSED MATERIAL, EVEN IF THE LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH LOSSES, COSTS, EXPENSES, OR DAMAGES. WHERE A LIMITATION OF LIABILITY IS NOT ALLOWED IN FULL OR

IN PART, THIS LIMITATION MAY NOT APPLY TO YOU.

c. The disclaimer of warranties and limitation of liability provided above shall be interpreted in a manner that, to the extent possible, most closely approximates an absolute disclaimer and waiver of all liability.

Section 6 -- Term and Termination.

a. This Public License applies for the term of the Copyright and Similar Rights licensed here. However, if You fail to comply with this Public License, then Your rights under this Public License terminate automatically.

b. Where Your right to use the Licensed Material has terminated under Section 6(a), it reinstates:

   1. automatically as of the date the violation is cured, provided it is cured within 30 days of Your discovery of the violation; or

   2. upon express reinstatement by the Licensor.

   For the avoidance of doubt, this Section 6(b) does not affect any right the Licensor may have to seek remedies for Your violations of this Public License.

c. For the avoidance of doubt, the Licensor may also offer the Licensed Material under separate terms or conditions or stop distributing the Licensed Material at any time; however, doing so will not terminate this Public License.

d. Sections 1, 5, 6, 7, and 8 survive termination of this Public License.

Section 7 -- Other Terms and Conditions.

a. The Licensor shall not be bound by any additional or different terms or conditions communicated by You unless expressly agreed.

b. Any arrangements, understandings, or agreements regarding the Licensed Material not stated herein are separate from and independent of the terms and conditions of this Public License.

Section 8 -- Interpretation.

a. For the avoidance of doubt, this Public License does not, and shall not be interpreted to, reduce, limit, restrict, or impose conditions on any use of the Licensed Material that could lawfully be made without permission under this Public License.

b. To the extent possible, if any provision of this Public License is deemed unenforceable, it shall be automatically reformed to the minimum extent necessary to make it enforceable. If the provision cannot be reformed, it shall be severed from this Public License without affecting the enforceability of the remaining terms and conditions.

c. No term or condition of this Public License will be waived and no failure to comply consented to unless expressly agreed to by the Licensor.

d. Nothing in this Public License constitutes or may be interpreted as a limitation upon, or waiver of, any privileges and immunities that apply to the Licensor or You, including from the legal processes of any jurisdiction or authority.

==========

Creative Commons is not a party to its public licenses. Notwithstanding, Creative Commons may elect to apply one of its public licenses to material it publishes and in those instances will be considered the “Licensor.” The text of the Creative Commons public licenses is dedicated to the public domain under the CC0 Public Domain Dedication. Except for the limited purpose of indicating that material is shared under a Creative Commons public license or as otherwise permitted by the Creative Commons policies published at creativecommons.org/policies, Creative Commons does not authorize the use of the trademark "Creative Commons" or any other trademark or logo of Creative Commons without its prior written consent including, without limitation, in connection with any unauthorized modifications to any of its public licenses or any other arrangements, understandings, or agreements concerning use of licensed material. For the avoidance of doubt, this paragraph does not form part of the public licenses.

Creative Commons may be contacted at creativecommons.org.