

Chapter 7

The Open Source Process

Failure and Success

Why are some open source projects—such as the KDE Project or the GNU C compiler gcc—highly successful, whereas others miserably and/or silently fail? From years of observing and participating in the open source world, we believe that more projects might fail than succeed. Of course, hardly anyone sends out press releases saying, "The FooBar Project Officially Declared Dead." Instead, the project home page stops getting updated, announcements of new releases stop appearing on mailing lists, and if you contact the maintainer, he or she says something like, "Oh yeah, we haven't touched that code in months. Go ahead and see what you can do with it, though." Software, like old soldiers, never dies; it just fades away.

All projects begin with the expectation of success and the hope that large numbers of hard-working developers and enthusiastic users—some of whom will contribute bug reports and patches—will immediately adopt the software. In most cases, the project finds a kind of comfortable middle ground: A modest number of users grow to depend on the software, they find each other and band together, usually on a mailing list or newsgroup, and stay in close contact with the maintainer or maintainers. Some of them are able to help out with debugging, creating new features, and generally keeping the code healthy. This is what we like to call a "fireside user community"; every program needs one if it is to stay alive. Although software might be temporarily sustained by artificial means (for example, influxes of money or unusual dedication from a single programmer who is emotionally attached to the code), a program is doomed to extinction unless it has some committed

users who depend on it to get things done. Many projects that appear initially promising end up as failures—that is, they don't spread beyond their original authors—because they don't fulfill this basic requirement.

A basic principle of the open source world is that software, if it is to survive in the long run, needs to fulfill a well-defined and acute user need. As Eric Raymond stated in *The Cathedral and the Bazaar* (**www.tuxedo.org/~esr/writings/cathedral-bazaar/**), his open source program fetchmail survived and thrives because many people need this piece of software to get their email to their local mail system when they connect by modem to their ISP. This is a pretty narrowly defined requirement and fetchmail does fulfill it. And that is why it is successful.

Successful projects also tend to have certain organizational qualities in common; the failures have certain structural flaws in common, as well. Doubtless, every project's fate is also partially determined by its unique circumstances, but the outlines of some general principles of free software development have become clear in the last year or two. In this chapter, we'll try to explain how to run a free software project so it survives.

Did warning bells go off in your head just now?

We hope so. Open source development is still a very young field; anyone making authoritative pronouncements about it should be regarded with extreme suspicion. Humility—or at least common sense—compels both authors of this book to issue a few disclaimers.

First, remember that free software is still in early stages where a wide variety of ideas and methods and experiences proliferate, because the unsuccessful variations have not yet had enough time to fail convincingly. Indeed, the entire field of computer programming is still a bit fresh when measured against older, better-understood crafts such as building construction or chemistry experiments. No mere human intelligence can compete with an evolutionary process when it comes to distinguishing good practices from bad ones, yet that is what we must try to do in software development, with far too little history to guide us.

Second, "success" means different things for different projects. Sometimes a "failure" at least accomplishes an educational purpose, or perhaps it serves as a useful stopgap measure until some other, more complete or robust piece of code is developed to replace it.

Most software projects start out with roughly the same goals, though: to solve a particular problem and to acquire users. As in many endeavors, starting off on the wrong foot can seriously reduce the project's chances of meeting either of these goals, so first we'll consider the most fragile stage of every free project's life cycle: the beginning.

Starting a Project

As we said before, if the code doesn't have a user community committed to keeping it alive, it will eventually fade away. But clearly, no program starts out with dedicated users from day one. The code has to be reasonably stable and actually do something before users will even take the

time to try it out. How does it get to that stage? This is the chicken-and-egg dilemma faced by every new project and is probably where an important number of fatalities occur.

The terminology and principles Raymond introduced in his book *The Cathedral and the Bazaar* have become so widespread that they are a good starting place for examining many of the principles of free software development—although there is room for debate on certain of his assertions. At any rate, his take on why projects get started is highly persuasive: *Every good work of software starts by scratching a developer's personal itch*.

There are exceptions, but in general, new free software is written because someone wants to see a certain problem solved. The reward comes from the convenience of having code do what humans formerly had to do, not from selling copies of the software. Therefore, if the project is to sustain the interest of even one programmer for long enough to reach usability, the project goals must be kept in clear view the entire time, and steady progress must be made throughout the entire initial development stage. The larger and more significant the goals, the longer the project can sustain itself on promise (known in the industry as "vapor") before it must show results. However, if the project doesn't show results within a finite amount of time, people lose interest and move on to another project that has more obvious potential.

Thus, if anyone, including you, is to care about the code enough to stick with it, he or she must first care about the problem the code was written to solve. This is usually not too difficult—after all, you wouldn't have started writing it if you didn't need it for something. The question is, will anybody else be interested?

More than likely, the answer is "yes." And that is because usually one's most specialized needs turn out to be shared by tens or hundreds of other people. Your "personal itch" is, most likely, not unique to you. The phenomenon of *parallel ideation* has long been observed in the older arts and sciences; a certain idea will somehow be "in the air," albeit unspoken, and people independently begin to notice and act on it—all at the same time and without being aware of each other's work. This phenomenon happens in software design, too. With significant statistical frequency, free software projects designed to solve the same problem will spring up simultaneously in different places. Often, whichever one gets the most success early on wins by drawing resources away from the others. Because outside contributors are free agents, they turn their attention to whichever code base seems likely to solve their problem first.

This process has an obvious implication for you as an initiator of a project: You should look to see if anyone else has already started something similar. Where you look depends largely on the subject domain; you'll probably know the appropriate newsgroups, mailing lists, and Web sites to check. Some good generic resources are Freshmeat, a Web site specializing in announcements of new software (**www.freshmeat.net**), the Ask Slashdot archives (**www.slashdot.org**), and the Free Software Foundation's software links (**www.gnu.org/software/**).

If (as is quite probable) you discover a similar project already under way, you might be better off joining it than starting your own project. It's a matter of determining how closely their goals match yours, examining what they have so far, and possibly asking questions of the authors. If their purposes are very close to yours and they already have running code, you can become a contributor (some work) instead of a primary maintainer (lots and lots of work) and still receive all the benefits of the package.

If you don't find any similar work already under way, the next step is to put your project in public view as soon as the time is right, probably in the very same forums you originally scoured for news of other projects.

But when is the time right?

Think about it from a different perspective: Is there any reason not to tell others about your code? There is strong pressure to announce to the whole world the moment you start coding. After all, the sooner they know, the sooner they can help out. And, just possibly, the sooner they can stop working on their own half-begun projects to do the same thing. Right?

Wrong.

It is better to resist that pressure. Openness and sharing are wonderful things, but attention is still a nonrenewable resource, and you're in danger of wasting other people's time if you attract them to your project before it is ready. Your reputation is on the line. By telling interested parties that you've started work on a certain program, you've given them the right to expect some running code to play with. If all that is available is a README file, a few rough notes on design, and maybe some skeletal code, these possible contributors will realize there's nothing you've done that they couldn't do themselves in a few hours. Perhaps more devastating, they will take you less seriously the next time you make an announcement.

On the other hand, if you wait too long, you run the risk of someone else announcing that they have running code that does just the same thing as yours. At that point, you'll have to decide whether to throw away everything you've done so far or continue in open and friendly competition with the other project. There's another danger to continuing too long in isolation, as well: You might miss out on some great contributions—contributions that would have affected the overall design of your code if received early enough. And design errors, as everybody knows by now, are the ones most likely to kill your software later.

Release Something Useful

Lest you think all these possibilities exist only in some abstract, theoretical world, here is some actual history from a project of one of this book's authors, Karl Fogel. In March 1999, he released a small program to the Net. This quite specialized program reformatted the output of the **cvs log** command into the much more readable GNU ChangeLog style (covered in Chapter 10). Although he wasn't completely satisfied with it at the time of release—it lacked several important features—it did perform its basic function without crashing or complaining, and he had a hunch it was something other people would find useful. So he

posted it to Ask Slashdot and to an appropriate mailing list (in this case **info-cvs@gnu.org**) and waited for the bug reports and feature suggestions to flow in. They started arriving almost immediately, including the following from Melissa O'Neill (**oneill@cs.sfu.ca**):

```
Hi Karl,
In the spirt of free software, here's a patch to your program to handle
the non-atomicity of CVS checking better.
Best Regards,
Melissa.

[patch followed]
```

Her patch, although introduced with little fanfare, actually contained a thorough rewrite of the program's already complicated main loop. It was just the sort of change that you would want to make early in development, rather than later when a lot more dependencies might be affected. Furthermore, in another email, she wrote:

```
From the first time I downloaded CVS, I couldn't believe this support
wasn't there. I resolved to one day write a Perl script to turn cvs
log into something useful, but you beat me to it...
```

The "shared itch" theory had held up wonderfully—not only was the program what she had needed, but she had even been considering writing it herself! Fortunately, she hadn't done any significant coding yet and was able to get everything she needed from Karl's program—and what it lacked, she simply added herself. Over the next week or two, Karl received 19 separate patches from Melissa, all of which resulted in definite improvements to the code. There were also numerous bug reports, patches, and suggestions from other people (15, to be precise, 14 of whom were complete strangers to Karl). As a result of all this activity, the program progressed at a far faster rate than would have been possible had Karl continued working in isolation. Bugs were found and fixed, algorithms improved, and features added—yet Karl's role was mostly to coordinate the changes that others sent in. This was strong confirmation to Karl both that the program had found its niche—its fireside user community—and that he had released it at the right time—when it was working but still needed improvement.

The best guideline for an initial release, then, is to post the code when it's far enough along to reliably solve some significant portion of the problem for which it was designed. It should do something useful right away, even if it doesn't yet do everything it's intended to do.

The key word is "reliably." If the program still crashes occasionally, find out why and fix it before your release. You might be anxious to release right away, but delays spent fixing showstopper bugs are really worth the time. Nothing impresses potential users (and contributors) more than rock-solid stability and graceful handling of error conditions—and nothing scares them away like unexpected crashes. All the glitz and flashy features in the world won't mean a thing if the code's foundation is visibly shaky. (We call this the "just

catch it" principle, from a more experienced juggling partner who, on seeing unnecessarily fancy receives attempted and consistently dropped, said, "That's nice, but you know, it always looks better if you catch it.")

The majority of potential contributors often feel that they can patch in a desired missing feature themselves, but that if the fundamental code is fragile, there's no point in their putting new code on top of it. Even the most charitable reaction to crashes is, "Well, I guess it's not really ready for outsiders yet. I'll come back in six months and see what the code looks like."

The importance of releasing runnable code was demonstrated by one of the most well-publicized mis-starts in the history of free software. Netscape Communications Corporation announced in January 1998 that it would publicly release the source code to their Navigator Web browser under a license allowing others to modify and redistribute the source code. The news sent a wave of excitement through the free software community. The lack of a good free Web browser had long been a serious problem, and Netscape was claiming that it would solve it in one stroke.

Netscape's sincerity was not faked; the company did exactly as it said. However, three and a half years later, the project is a failure by almost any measure. It is still possible that a working Web browser might arise from the tangle of code available from **www.mozilla.org**, but don't hold your breath. Despite the huge demand for a free Web browser and the many programmers willing to contribute their talents to make it happen, the long-awaited production release of Mozilla still does not exist.

What did Netscape do wrong?

Ex-Netscape employee Jamie Zawinski has analyzed the causes of the collapse in some detail at **www.jwz.org/gruntle/nomo.html**, a fascinating essay that offers several reasons for the failure. We're willing to bet, however, that his Reason Number 2 is the true cause: Netscape's initial release was not running code.

Instead, Netscape released the latest snapshot of its development tree, or as Zawinski put it, "What we released was a large pile of interesting code, but it didn't much resemble something you could actually use." Developers were not treated to an "out of the box" experience; instead, they had to struggle just to get the code to compile, let alone run.

This problem alone might not have been fatal, as long as making Mozilla run was the only way some people could get a working Web browser. But everyone already had a Web browser—usually either Netscape Navigator or Microsoft Internet Explorer. These browsers might not have been open source, but they worked well enough and didn't cost any money. So the developers' main "itch" was already being scratched. Thus, all the people who would otherwise have worked feverishly to ship a working version of Mozilla right away instead felt they had the luxury to turn Mozilla into a textbook-perfect Web browser. In retrospect, it is all too obvious—no one was really depending on Mozilla to get anything done, so actually shipping it became an infinitely receding priority.

It was a measure of Mozilla's promise, and the intensity of the community's desire for a free Web browser, that many people still contributed. Unfortunately, in software, there is no strength in numbers. The programmers were there, with plenty of technical ability, but they had no compelling reason to make the code work right away; their efforts never focused enough to get the program to run.

It was a costly and probably embarrassing lesson for Netscape, but at least now the rest of us can learn from it: running code—first, last, and always.

Packaging

Getting the program to work is only half the story, however. Although the code itself is the main attraction, the packaging you wrap it in can make a big difference in initial levels of participation.

Packaging means at least some basic documentation, probably a project home page on the Web with a clear description of what the code does and what's required to run it, and a convenient way for the public to download the latest updates to the code. Too often, programmers treat these things as mere decoration and don't devote enough time to providing them. Perhaps it's because of the legendary programmer's distaste for anything that feels like marketing or public relations, or perhaps it's due to a loss of perspective. After all, when you know your own code front to back, it can be difficult to comprehend that others might need introductory documentation before they can use it. Whatever the reason, bad packaging can seriously detract from a project's attractiveness, which is a shame when the code itself is of good quality. Don't let your project falter because of an unnecessarily steep learning curve for new participants.

The first piece of packaging is the license under which you distribute the code. It might seem trivial, but people can't be confident that your program is really free software until they see its actual terms of copyright.

Free licenses tend to fall into two groups: the GPL (GNU General Public License) I and II, and all the others. The GPL (printed in Appendix A of this book) insists that not only the code itself, but also any derivative works, must be distributed under the GPL; thus, it is self-perpetuating and infectious. Other licenses state that the code is free as received but allow derivative works to be distributed under different (and sometimes more restrictive) terms, as long as proper credit is given.

Occasionally, people even write an entirely new license to accompany their software release, designing the license to embody everything they think free software should be. (This seems to be the course favored lately by large corporations that have suddenly decided to release code to the public, such as Sun Microsystems with its Solaris operating system.) Writing a custom license is very definitely *not* recommended; for one thing, it's hard to do it right. Did you use a lawyer? Did you make sure the lawyer really understood your goals in releasing the software this way? Worse, it's simply tiresome for potential users to have to read and evaluate unfamiliar licenses all the time. Unless you really do have special needs, just stick with one of the standards.

Licensing choices are the subject of fearsomely intense debate among free software advocates, and we are simply not going to get into that debate in this book. We like the GNU GPL license. We hope you'll use it for all of your programs, but even if you don't, it probably doesn't matter very much. It is not the details of the license that keep the code free, but the fact that free code has better survival characteristics than nonfree code. Even if for some reason you choose to distribute your program under a license allowing nonfree derivative works, those nonfree descendants are probably doomed to have a shorter life span than the free ones. So don't worry about it too much. Just pick a license that ensures everyone else the same rights as you have with respect to the code, or put your code in the public domain if you can't make up your mind, and let the dynamics of free software do the rest.

The rest of the packaging tends to involve more mundane matters of presentation. Here's an informal checklist we use before releasing new code. We include it here not as a rigid standard, but to give you an idea of the sorts of things that make a project developer-friendly as well as user-friendly.

- At least one person other than its author has tested the code. Everyone knows that bugs tend to hide from authors, only to reveal themselves as soon as someone else tries the code. Impose on a friend or colleague to give your code at least a cursory test run before you post it to a bunch of strangers on a mailing list or newsgroup.

- Documentation is available in an obvious place. It doesn't have to be much at this stage, just a quick summary of the purpose and basic usage of the program. If the project is not overly complex, the documentation need not even be a separate file—it could just be the output from invoking the code with a **-help** option or something like that.

  Also, a special plea: If the documentation is in a separate file, use a widely understood, easily searchable format such as plain text or HTML. If you also want to offer PostScript, RTF, PDF, and that sort of thing, that's fine, just as long as those aren't the *only* ways to view the documents. It's possible to alienate an astonishing number of developers in a short amount of time by requiring them to print out a hard copy or download special viewing software just to read the documents. Most of them will depart in frustration rather than continue, reasoning that if the first thing they encounter is so developer-hostile, the rest of the project is likely to be, too.

- A project home page or some other central place exists where people can find up-to-date information about the project (a Web page is the norm). It introduces the program by first describing the problem that the program was written to solve (so visitors will be able to determine quickly whether they're interested in reading more), then summarizes how the program solves it. The page also lists any unusual system requirements that people may need to run the program and offers links to the documentation, the source code, and any other applicable materials. Finally, the page makes it completely clear that the program is free software, by using the words "free software" or "open source", and by linking to the actual copyright.

- The latest sources can always be retrieved from a clearly designated location. For the latest released version, you can link to it from the home page. For those who need access to the continuously changing development sources, the answer is, of course, CVS. Making the code accessible via a public CVS server allows early enthusiasts to become as involved as they want to be, giving them access to code changes right away instead of making them wait for official releases. (The details of setting up a CVS server to allow anonymous, read-only access are covered in Chapter 3.)

  Most users, however, will use the latest RPM (Red Hat Package Manager) versions, if any, posted in public forums, because they are the most convenient alternative and because such posted versions are usually thought to be stable and have an implied endorsement from the author. Those who are interested in following the code closely or perhaps in becoming developers themselves will want and expect continuous access to the latest sources. Access to development sources is also crucial for bug reporters, who will usually update to the latest version of the code and try to reproduce the bug again before they send in a report that is relevant.

- The initial release and every release thereafter has a clear version number. This might seem obvious, but you'd be surprised at the number of programs—especially smallish scripts—that are posted without a version number.

  The purpose of the number is *not* to mark psychological or marketing milestones in the code's progress (although it is traditional for release 1.0 to signify a stable, workable product, release 2.0 to be a major improvement over that, and so on). The real purpose of the version number is to give people an easy, unambiguous way to compare two versions of the code and to know which is more recent. To that end, it does not matter if the version numbers convey any information about the degree or significance of the changes between them. Although the difference between releases 2.0 and 1.0 is usually greater than the difference between releases 3.90.8 and 3.90.7, this is not a requirement. However, there is a requirement that release 2.0 be newer than 1.0, and release 3.101 be newer than 3.99. This consistency is vitally important: When users or developers decide to upgrade to the latest version, they must be able to compare their current version with what's available on the project home page and know instantly whether there are any newer versions available.

Note

Although CVS can increment version numbers for you automatically, this is not necessarily always desirable. The pros and cons of automated version bumping, as well as the mechanism, are covered in Chapter 6.

- An email address has been designated for reporting bugs. Although not strictly necessary, we've found that it's often a good idea to separate the bug address from your personal address. When additional developers start to join, you can just add them to a communal bug list, instead of manually forwarding bug reports to them.

A non–bug developers' mailing list can be helpful, too, but you'll probably want to wait until you have several developers trying to communicate with each other. If the project is large and likely to stimulate a lot of early developer interest, you might prefer to have this list address ready before you announce your software.

Don't be surprised if setting up all of this takes as much time as actually writing the code, or perhaps even more. Even though the packaging work might not feel as productive as coding, it will pay off quickly when people join the project and make the pleasant discovery that an infrastructure has already been established. This will give prospective contributors confidence that the project is well managed, which in turn makes them more likely to contribute because they will think their contributions are likely to survive. The more steps you can take to inspire that sort of confidence, the better off the project will be.

Announcing the Program

After you've gone through the checklist and taken care of whatever packaging is appropriate, you're ready to post an announcement to the relevant mailing lists and newsgroups. The convention for such announcements is to have a subject line that looks something like this:

```
Subject: ANN: Genie 1.0, program to take genotype and print out phenotype
```

It's quite important that the subject line be accurate and concise, because most people just scan the subjects of messages in mailing lists and newsgroups and delete everything that doesn't grab their attention.

Make sure that the body of the announcement describes the project as clearly and quickly as possible and mentions the project home page early on, so interested parties can visit it immediately and bookmark it. If the program consists of just one or two small files, you could include them directly in the announcement message; otherwise, assume that anyone who wants the code will make the trip to the Web site.

That's it. Once the announcement is posted, you can sit back and wait to receive bug reports, feature suggestions, and help requests. The project has now entered its second and longest phase: maintenance.

Running a Project

You are now the maintainer of an active free software project.

A maintainer's job is to say yes or not at the right moment and to help others to code for the project and improve the software. However, in the vibrant OpenSource community a maintainer's job is most usually to say not at the right time to the right people in the right words. In any project with multiple participants, there's no way to avoid occasional disagreements over design, features, or sometimes even the program's very purpose. By accepting ultimate responsibility for the state of the code, the maintainer naturally ends up assuming

a degree of authority as well. (For some projects, the maintainer position is held equally by a group of people; this scheme is discussed later in the chapter. However, most projects start out with a single person in this role, so that's the situation we'll examine first.)

When the maintainer's decision is to say "yes"—that is, to accept a particular patch or a proposed code change—there is usually no great controversy. Although some developers might feel that the change is for the worse, they usually accept it. After all, the change probably benefits someone, or it wouldn't be proposed. It's difficult to make a forceful argument that some new feature or behavior is undesirable merely because it doesn't fit with the program's overall design.

However, when the maintainer must say "no," care and sensitivity are required. Rejecting a contributor's code or idea is never easy—especially when someone has spent a long time implementing and testing the change—but it must be done occasionally if the program is to stay healthy. Good quality control means saying no; the only question is how to reject undesirable contributions in a constructive manner.

Luckily, the maintainer starts out with a certain amount of moral authority, and it is generally accepted that the maintainer has the final word in disagreements. In the absence of a more formal decision-making structure, other developers tacitly accept that the maintainer is a benevolent ruler. When no clear consensus can be reached on an issue, the maintainer simply decides by fiat, and that ends the matter. (Well, it rarely ends the discussion, but at least it settles the question of which direction the code will go.)

This might seem surprising for a system that boasts of its democratic and decentralized nature. However, great software, like great art, is rarely created by committee. To resolve disputes quickly, the most efficient way (note that "efficient" does not always mean "best") is to have a designated arbitrator—in other words, the maintainer.

The crucial factor that makes this arrangement acceptable to developers is that the maintainer's authority is based on merit, not ownership. If people disagree strongly enough with the maintainer's decisions, they can make a separate copy of the code and start distributing their own divergent version of the program, one in which their decisions are implemented. This is known as "forking" the code (or sometimes "branching," not to be confused with a literal CVS branch).

A fork is a very serious event. It causes confusion among the program's users, who now have to decide which fork of the program is more likely to meet their needs and to survive in the long term. Moreover, a fork can give rise to long-lasting political divisions. As a practical matter, most developers must choose which version to devote their time to (although there are always a few developers who simply spend the extra effort to make their contributions fit both forks).

Fortunately, forks are quite rare, partly because they're so much trouble to undertake, but mainly because the implicit threat of one occurring is usually enough to prevent the maintainer from making decisions that upset a majority of the other developers. Avoiding the

possibility of a fork is a strong incentive to be a truly *benevolent* ruler. Your subjects can duplicate your kingdom at any time and remake it in their own image, so you had better pay attention to their opinions!

Benevolence is not just a state of mind; it has quite specific implications for how a maintainer ought to behave. First—and most important—you must always explain the reasons behind your decisions. People often ask a maintainer to accept or reject an idea based only on a vague description. But without an actual patch, or at least a highly technical description of the change, it's usually impossible to determine its worth.

When the change is not too big, it's not necessarily rude to ask someone to actually implement it before you decide whether it's worth folding into the code. After all, working source code can be viewed as merely the last in a series of increasingly precise behavioral descriptions; to ask for source code is to ask for the most exact possible description of a change, which is sometimes the only way to know if it's worthwhile.

However, if a proposed change is going to require many hours of work, it's understandable that the person wants to know in advance that it will be accepted. In that case, most maintainers typically work with the contributor to produce a specification that resolves all the important questions about the nature of the change, while still reminding the contributor that the patch must be reviewed prior to acceptance. This might seem overly harsh, but most developers understand that a maintainer cannot judge a patch without actually reviewing and testing it. Of course, if the maintainer ultimately rejects the patch, he or she must give the contributor specific reasons.

When rejecting a patch or a proposal, however, the maintainer cannot merely explain why; he or she must explain the reasons politely and without rancor. A contributor's proposals and opinions must be given respect and at least the appearance of consideration when discussed in public forums. Criticisms do not have to be suppressed, but they must be specific, articulately argued, and never personal.

Maintaining a high level of civility while still giving good critiques not only increases the likelihood that the developer will eventually contribute something useful, it also encourages others to contribute, simply because the overall environment will be conducive to high-quality debate, free of personal attacks.

However, it might happen that sometimes a maintainer is so overwhelmed with work that the good tone suffers for a short while. A good maintainer must spend as much time on organizational and social issues as on coding. This might sometimes be a bit of a mismatch with a maintainer's actual skills, as it is well known that the best coders (and, therefore, the ones likely to get code up and running fastest) often devote a disproportionate amount of their minds to programming, leaving correspondingly less available for the niceties of human relations. If you think this describes you, our best advice is to make a conscious decision to read every email and posting that you send out as though you were going to receive it

yourself. Like any skill, managing projects gets easier with practice. Most participants don't expect an ambassadorial mastery of high diplomacy; they just want to be treated with the respect due any volunteer.

Tip

While we're on the subject of making a project developer-friendly, it should be noted that Internet-based projects often attract contributors from many different countries. Miscommunication is rare, fortunately; it seems that English has become the language of international software development, and most programmers know enough English to participate without too much trouble (usually even to the point of being fluent in computer jargon). Whether this is a good thing or not is beyond the scope of this book. Esperanto isn't exactly taking over, and a lot of code and documentation is already written in English. For better or worse, most people seem to be following the path of least resistance. However, don't fall into the trap of forgetting who is on a developers mailing list or newsgroup and make a remark that is only comprehensible (or even appropriate) for people from your own country.

Cultivating Technical Judgment

Now that you know how to tell someone diplomatically that his or her change won't be accepted, how do you decide whether to accept it? This is not merely a matter of having the requisite technical judgment, but of demonstrating it in such a way that both you and others are *confident* that you have it.

The answer, perhaps surprisingly, is not to try to be the biggest frog in the pond. You do not need to be a programming guru with vast experience and imposing credentials; indeed, you might well recognize that many of your contributors are far more experienced coders than you are (we've certainly felt that way many times!). What you do need are two things: a clear sense of the program's purpose, and the ability to recognize maintainable solutions.

The first is as much a matter of knowing what the program should not do as knowing what it should. Curiously, when designers err in imagining the scope of their projects, they tend to err on the side of inclusiveness rather than restrictiveness. It's extremely tempting to concede no limits, especially early in development; the less actual code that has been written, the wider the apparent possibilities for future development.

Unfortunately, a program that does everything probably doesn't do it very well. The best programs, free and otherwise, know the problem they were written to solve and succeed by addressing every aspect of that one problem. Free software starts out with an advantage here; most free programs were developed because someone wanted to solve a specific problem. When the program later expands beyond that, as it generally will, it's usually because people found a related problem that frequently needed to be solved in conjunction with the original one. (Commercial software, on the other hand, is just made to sell as many copies as possible; this often leads the maintainers to pile on as many features as possible so that

there is something for everyone.) A clear understanding of the program's goals, combined with a vigorous fear of software bloat, will aid the maintainer in deciding which changes should be accepted and which shouldn't. Furthermore, that understanding will make it much easier to justify decisions to others.

This need to articulate the program's purpose early in its life might feel overly confining but, in the long run, it is really healthy. This statement of purpose doesn't have to be absolutely restrictive; it can evolve over time, in response to unexpected uses. Just such an evolution happened with CVS itself, in fact. Originally, CVS was regarded as only a tool for allowing a group of preselected developers to collaborate. One day, someone combined two of its features (the remote server and read-only repository access) to allow random strangers to check out noncommittable working copies anonymously. As soon as everyone else realized how useful this was, CVS became both a developer-collaboration tool and a software-distribution tool. The expansion was so natural that virtually no objections were raised, and it's safe to say that decisions about CVS's direction will take both aspects into account from now on.

Note that real-world usage—not the maintainer's or developers' ideas of what people might or should want—drove each of those features. Both the remote server and the read-only repository functionality (which together made anonymous access possible) were implemented because of immediate demand—the code was written to fulfill a need, not the other way around. "Evolution," in the context of free software, is more than just a buzzword. It means that the maintainer is guided by how the program is used in the real world; he or she doesn't try to second-guess reality when deciding what to implement. Even when a program's domain (the range of problems that it can be used to solve) expands, it rarely expands into completely unexpected territory. The border is pushed gently and incrementally, so that at no point does anyone—developer or user—wonder why a particular feature was added or behavior changed.

Thus, the questions to ask yourself when considering whether to implement (or approve) a change are:

- Will it benefit a significant percentage of the program's user community?
- Does it fit within the program's domain or within a natural, intuitive extension of that domain?

Failure to satisfy one or both of these conditions is reason enough to reject a change. The appropriateness and usefulness of the new functionality is not merely a matter of aesthetics; it has a direct bearing on maintainability. A stretch of code that doesn't get regularly exercised will inevitably suffer bit-rot. Therefore, it's a losing proposition to add a new code path unless you can be confident that it will be executed often enough for any bugs to be found and reported. If no one but the feature's contributor wants or expects a given new feature, that feature will never be widely enough used to become truly robust.

As far as we know, there's no standard ratio relating the number of changes accepted to the number proposed. You must use your judgment. However, you can and should consult with other developers, either on an open discussion list or via private email, when considering a change. In fact, in the most successful projects, the maintainer seems to engage in such consultations regularly. Most developers view a maintainer's inclination to seek out all points of view on a given change consistently as a sign of confidence on the maintainer's part. No one expects a maintainer to be an infallible design guru or even to always make inarguably right decisions. People *do* expect a maintainer to be sensitive to the community's opinions, even when he or she must oppose them, and to provide thorough justification for controversial decisions.

Having clarified in your mind that a particular change is a good idea, you still face the question of whether a contributor's patch to implement this change is technically acceptable. There is no magic formula to answer this; you just have to roll up your sleeves and dive into the code. It's normal for a maintainer to become quite skilled at reading patches and quite all right for the maintainer to request that diff output files be sent in the format that he or she finds easiest to read (most likely either context or unified format—respectively, the **-c** or **-u** option to **diff**).

Reviewing patches is not easy. Most patches need at least a little massaging from the maintainer. The mere fact that a patch doesn't apply flawlessly on the first try or even contains a few buglets is probably not enough reason to reject it outright. Instead, you must look over the patch line by line, with the original sources at hand for reference, and decide if it works overall. If it does, you can apply it and manually clean up any loose ends (we give a detailed description of this process in Chapter 6).

It's reasonable to ask your contributors to abide by coding conventions, as long as the demands aren't too burdensome. Two common conventions are:

- All patches must be accompanied by log entries (summarizing the changes and making clear what parts of the code are affected).
- The changes must adhere to the same indentation standards as the rest of the program.

These are a small effort for the developers and greatly reduce the maintainer's "comprehension overhead" when looking at new code. If you can't easily tell exactly the effects of a change, immediately ask for help, not only from the contributor but from other developers as well. Often, they'll be able to point out things about the change that haven't occurred to you. The great strength of free software, as Eric Raymond said, is that "Given enough eyeballs, all bugs are shallow."

So, Who Is the Maintainer, Really?

Throughout this discussion, a tiny logical inconsistency has been left dangling: We might agree that the maintainer retains his or her position by demonstrated merit, but how is the first maintainer chosen?

The obvious answer is the correct one: In almost all cases, the original author of the software, the one who initially published it to the world, is the first maintainer. Because the first author (or authors) demonstrated merit by developing the software, in the absence of any adverse indications, people accept this original version of the program as the "canonical" version. In other words, if everyone is downloading the version of the code that you post, that must mean that you're the maintainer!

Thus, the best evidence that someone is the maintainer of a free program lies in the willingness of others to agree that it is so. This is in direct contrast to the world of commercial software. Microsoft, Inc., is the "maintainer" of MS-Word because it has an exclusive and restrictive copyright on the code. No matter how its users (or even employees) feel about its stewardship, Microsoft will continue to be the maintainer because it has a permanent legal right to the position.

Rule by Committee

Up to this point, we've been talking as if the maintainer is always a single person. Although most projects start out that way, when a project grows large or important enough, it's not unusual to have a group of developers accepting equal responsibility for the code. Such groups are usually structured, at least nominally, as self-regulating democracies (although in practice, one or two "senior" developers' opinions often informally carry extra weight). Typically, the existing developers vote on new members, and the properties that distinguish official developers from the rest of the world are:

- CVS **commit** access to the repository
- Voting rights (for example, voting on code changes or whether to accept new members)

There's no simple rule defining when it's appropriate to move to this kind of development model. Free software users often feel that once a program has become important to a lot of people, it's not quite proper for everything to depend on one individual. Even when the primary maintenance continues along the benevolent dictator model, it's not uncommon for assistant or backup maintainers to be explicitly designated. (Eric Raymond has done this for the fetchmail program he maintains, and Linus Torvalds seems to have done the same for the Linux kernel.)

CVS itself is group-maintained and even has a description of the development process (**www.cvshome.org**). Interestingly, the process for approving changes is not heavily formalized; decisions are made by informal consensus, and generally any of the developers can check in contributed changes. Existing developers propose and vote on new developers; sometimes developers resign from the group because they've become too busy with other things.

By contrast, the Apache Group (maintainers of the extremely popular Apache Web server) has a little more love for procedure. They vote not only on membership, but also on every change to the software; three positive votes and no negative votes is enough to allow a change to happen. How-

ever, even the Group's charter (see **www.apache.org/ABOUT_APACHE.html**) makes it clear that the system really depends on people exercising good judgment and having a healthy respect for the opinions of their peers, rather than on a precise and unambiguous written agreement.

These loose development charters are frequently mistaken for legal documents, but they actually serve a very different purpose. Legal constitutions are usually written with the assumption that the parties involved don't always necessarily want to cooperate, so that part of the constitution's role is to be a contract on which a neutral judge can base arbitration. Free software developer groups, on the other hand, are self-selecting for cooperators. After all, if someone doesn't want to cooperate, they're perfectly free to copy the code and start their own fork. Therefore, everyone in a given developer group is there because they want to be involved; there's no need for the charter to anticipate every possible misinterpretation. If disagreements do arise, people have every motivation to reach a consensus quickly, and they usually do.

Admittedly, dilemmas occasionally can arise from such loose systems. For example, when a developer group splits into two camps because of a major disagreement about some code change, how do you decide which side is the original and which is the fork? Both sides might feel equally legitimate guardians of the program, so each side wants to portray the other as having started a fork. Or what about two developers, both with **commit** access to the sources, who conduct an argument by continually reverting each other's changes—how can the rest of the group stop the cycle? If the two warring developers are equal with everyone else, then strictly speaking, no one has the right to take steps that bar either one of them from doing those things in the repository.

In practice, however, these things rarely happen. For one thing, the consequences of violating these mutually agreed-upon rules are not really very serious. The code won't disappear or stop working just because someone decides to take a step that's not within accepted strictures. A project's procedural regulations do not require the same strict obedience as, say, a nation's, because everyone involved in a project knows that if enough of the developers get disgusted, they'll simply go off and start developing the code on their own. If one developer gets out of hand, the rest are usually quite willing to indulge whatever extra procedural steps are necessary to solve the problem.

The method you ultimately choose for your project—benevolent dictatorship or ruling council—depends partly on which arrangement you're more comfortable with and partly on what your user community wants. If you're the kind of person who makes a good dictator, and no one seems to mind, then go ahead. If you are more comfortable sharing responsibility and can find others from your developer group willing to share it, then widen the throne. It might feel as if you're giving up power, but you're not; with free code, everyone in the world has exactly the same powers. The so-called maintainers really control only matters of convenience, such as giving the world an "official" place to download the latest release of the code, submit bug reports, patches, suggestions, and so forth.

Even the phrase "latest release" is slightly misleading. A publicly recognized, generally accepted source for releases of the software exists only because all of the interested and competent developers have coalesced around the same code base (it's in their interests to work together). The code itself is just bit patterns. Anyone can take those patterns, make new ones, and release them to the world, under any name they choose; it doesn't require special permission.

The arrangement that you encourage for running your project should be chosen based solely on what is best for the code. If you decide to share the maintainer responsibilities—or even hand them off to someone else entirely—you won't really be giving anyone any powers that they didn't already have. You'll just be making it easier for the developers to work together and improve the code. On the other hand, if you think the project will run better if questions are resolved quickly by one person, and you are confident that you are that person, by all means preserve a happy dictatorship. In the long run, the code itself will gravitate toward whichever method is most appropriate for it. If it becomes too large and complex for a single maintainer, you'll begin to feel overloaded and start hearing complaints from users that needed patches aren't getting applied fast enough.

How to Do a Fork, if You Absolutely Must

Even though forks are relatively rare occurrences, it's worth taking a look at how they should be (and generally are) conducted. Not only is it useful knowledge to have in an emergency, but it actually sheds a lot of light on the innately cooperative nature of free software.

First, think long and hard about whether the fork is really necessary. What do you hope to accomplish with it? If the problem is that bugs are not being fixed soon enough in the current version, you should devote your time to being a better bug-fixer within that development group. If the issue is really one of a major disagreement about the direction in which the code should go, a fork might be appropriate. Even then, though, you should ask yourself if you're positive that you can do a better job with the code than the current crew. If the answer is "yes," and you decide to go ahead with the fork, you must face the question of how to present it to the world.

Whatever you do, don't make an announcement that you're angry or frustrated with the current maintainers (even if you are) and that you're forking because you just can't stand it anymore. Instead, politely state the exact nature of the problems you have with the code's current state and apparent future, and what you intend to do (or better yet, have done) differently. You can post these announcements to the original development mailing list and on your "rival" project home page. You should also set up the forked project exactly as you would any other project, complete with a CVS repository, a bug-reporting address, a developer mailing list, and so on.

The most politically sensitive question is what to call your forked version. To decide that, let's step back a bit and clarify what the original maintainer can reasonably stake a claim to.

Although, as we have seen, the whole system rests on the fact that being a maintainer does not imply being an owner, sometimes a maintainer of a free program does own a legal trademark on the program's name. Even when this is not the case, the maintainer might rightfully feel that the reputation earned by the code so far was largely a result of his or her efforts and would be offended if other people began using the same name to distribute their rival code base. In a sense, this constitutes an implied trademark, to be honored until it has been proven beyond a reasonable doubt that the program has moved on to other hands.

There's nothing evil (by which we mean, of course, anti–free-software) about trademarks. Trademarks exist to prevent people from hijacking others' reputations and passing them off as their own. When a piece of free code is distributed under a certain name, that name is associated in users' minds with all the work that the maintainer and developer group have done up to this point. Because your concern is the future of the code itself, not the name under which it's distributed, your goal is not to get people to call your program by that same name right away, but merely to get them to use the code.

Therefore, the rule for someone considering starting a divergent branch is to only fork the code, not the name. If you run off and start distributing a rival version of program Foo, and you also call your program Foo, it will only cause confusion among users and make other developers suspicious of your sense of civility. If you have the slightest doubt about whether your fork is the diverging one, err on the side of caution and assume it is. It's only polite.

Of course, you want people to know that your new distribution is based on the software they've known from past releases and is applicable to a very similar problem domain. One solution is to give the new code a name that makes clear exactly what's going on. You can try "Rogue Foo," "Foo, Branch Edition," and so on, or use a mostly unrelated name but explain clearly in the distribution and on the project home page that it is a descendant of the Foo program.

Neither of these solutions is particularly elegant, but that's okay. The fork will either die out quickly or end up supplanting the new program. If it dies out, it doesn't matter that for a short while it existed with an awkward name. If it ends up taking over from the old distribution in the minds of the user community, and you're absolutely secure that that has happened, you can start calling it Foo again. After some time has passed, most people will forget that the fork ever happened, and any name confusion will be over. Remember, though, that the onus is on you to prove the new code's merit, so don't adopt the old name until the user community has made its preference for the forked version utterly clear. In cases where both the fork and its parent continue to exist side by side for a long time, eventually your program's name will acquire a reputation of its own, one that is associated in people's minds with precisely what your program actually does.

Choosing a nonconflicting name is one of the elements of a civilized fork, but not the only one. You should also remember that there is absolutely no reason to treat the original maintainer and his or her allied developers as enemies. If you think they're technically competent,

you should automatically make them full-fledged developers with **commit** access in the forked version. Whether or not they ever use the access, the message is clear. You are emphasizing that your goals are the same as theirs; your only disagreement is about how to achieve them.

Conversely, if you are the maintainer of a program from which a fork is diverging, there are also certain standards of behavior. No matter how you feel about the matter personally, don't criticize the instigators publicly. They came to their conclusions honestly, and although you might disagree, the user community will decide the issue of which version is better in the end. If the forkers had **commit** access to the original repository before, don't change that. Wish them luck and monitor their progress. If at some point you see that their version of the program really is doing a better job than yours, make a public statement that you now consider them to be the official maintainers, close down your version, and move on to other things.

In short, although a fork might seem like a hostile act, it does not have to be that way in practice. In fact, it would be contrary to the spirit of free software to make it so. It might seem like an unrealistically idealized view of human nature to think that people could cooperate in such circumstances, but in fact they do, because cooperation serves everyone better than competition. One of the most well known forks is the GNU Emacs/XEmacs divergence, now several years old, which occurred due to technical and organizational disagreements about how to produce the best possible text editor. Both versions are still going strong; all of the maintainers also state that they would like to see a re-merge happen someday, if the disagreements can ever be resolved. Most interestingly, the XEmacs team actively tracks changes to GNU Emacs and incorporates them into XEmacs. Meanwhile, on the GNU Emacs side, the maintainer (Richard Stallman) publicly admits that there's a lot of good code in XEmacs that he'd like to incorporate into GNU Emacs, if only certain organizational hurdles could be overcome.

This is not the behavior of fierce and implacable enemies; it's more like rival climbers ascending by different routes on opposite sides of a mountain, while trading information by radio and hoping to see each other at the summit soon.

Changing Maintainers

Not all maintainer changes happen because of a disagreement. Sometimes, someone just gets too busy, or too tired, to continue being responsible for a program. When that happens, it is incumbent on the maintainer to abide by yet another convention, well summarized by Eric Raymond: "When you lose interest in a program, your last duty to it is to hand it off to a competent successor."

You can advertise for a new maintainer on the current developer list; usually someone will come forward. If no one does, you can try posting a resignation announcement to the appropriate mailing lists and newsgroups, requesting that interested parties contact you about taking

over. From those who express interest, you should determine, as best you can, which of them is most likely to have the technical and organizational competence—and the time—to do a good job. The reason to take such care is that the world will very likely accept whomever you designate as the official maintainer; you don't want to let all those users down.

Of course, it's always possible that no one will volunteer. If you are determined to stop working on the code, and no one else shows signs of taking over, the code enters a kind of limbo state, where it is not being actively developed, but continues to be available. When asked about it, users say, "Oh, yeah, we still use it, but we think it's not being maintained anymore." The project has reached an end, although not necessarily *the* end.

Stasis

Free software projects never really disappear—not completely. Usually, there are too many copies of the code spread around the world for a given project to become literally unrecoverable. However, as the last known maintainer, you have a special responsibility to keep an archive of the latest version of the code, just in case someone with an eye toward restarting development ever asks you for it.

If possible, you should preserve the project home page and download links, with a notice stating that the project is no longer in active development and that potential new maintainers are invited to propose themselves. Whether you ever hear more about it depends on how important the code is to its users. As time passes, the code will rot, displaying funny behaviors or otherwise showing signs of age as it's run in newer and newer environments. Eventually, its users will band together to save it, if they really need it. If they don't, it was probably time to move on to something else anyway.

Knowing What We Don't Know

At the risk of repeating ourselves, we must state again that this account of the free software development process should be taken with a grain—maybe an entire shaker—of salt. We've tried to describe and analyze the way we've seen most successful projects work, but that doesn't mean it's the *only* way they can work. In addition to experimenting with the code itself, we hope you'll be encouraged to experiment with the *process* of running projects. One of the nicest things about CVS is that it functions as a safety net for your source code: It ensures that you can't lose any of the work you've done so far. That frees you and your co-developers to explore many methods of collaboration. When you find one that works for you, use it. It doesn't matter if it flies in the face of every piece of advice written here: If it works, it works. That is the final proof of any development method.