

Chapter 1

Why Open Source Development and CVS Go Together

What Is Free Software?

Traditional capitalism is based on the idea of limited supply; however, information has become a commodity in itself and is *never* in short supply. In fact, the ubiquity of computers and the Internet has made it possible to replicate any information effortlessly and without bounds. Even so, we still treat software as if it were a tangible object in short supply. If you copy software from somebody, you're legally stealing it. The software industry has attempted to extend this metaphor into the information economy, *artificially* re-creating the economics of limited supply by creating *software licenses*.

There's nothing wrong with making a living as a programmer or as a software company employee or executive. The authors of this book get part of their incomes as programmers. However, it's nonsensical to use this profit-model. Imagine a science-fiction device that allows any sort of food or physical object to be infinitely duplicated. If somebody then tried to sell you a tire for your car, why in the world would you buy it? You could just throw your friend's tire into the duplicator! However, you might want to pay somebody to *design* a new tire for you or perhaps to *install* the tire on your car. Or to help you when some other part of your car breaks, you might want to buy a warranty for future support. Or maybe just hire a personal mechanic.

Similarly, in a world where all software is in the public domain and infinitely reproducible, programmers and software companies are able to make a good living not by restricting the flow of software, but by providing a *service*. Users pay the programmers and

companies to design and write new public domain software, as well as install, maintain, customize, troubleshoot, and teach others about it. A programmer or company sells *labor*, not products—much like a mechanic, plumber, or electrician.

Getting back to the original question, then: *Free* means that the public domain software comes with *freedom*—its users have the freedom to use it however they wish, including copying it, modifying it, and selling it.

A particular company that is in the software business either directly, as an independent software vendor (ISV), or indirectly, producing software as a key component of other goods or services, faces several challenges. Among these challenges might be:

- Continuing to create new products and bring in new incremental revenue
- Improving new product quality at first release
- Doing a better job of sustaining engineering in supporting current and older releases while still driving innovation in new releases
- More effectively recruiting third-party developer and integrator support for the company's products and platform
- Motivating and retaining current employees and recruiting and energizing the next generation of employees

These challenges are interconnected for two reasons. First, most of them are functions of constrained resources: Few companies have enough people, money, or time to do everything that needs doing, especially when competing against larger companies with greater resources. Second, all companies like this have at least one available strategy that might help address all these issues together, turning some (or in exceptional cases even all) of your software products into "open source" products.

Open Source Software

You've no doubt read about Netscape's 1999 release of the source code for Netscape Communicator. You might also have heard about earlier open source projects such as the Linux operating system kernel or have read papers such as Eric Raymond's "The Cathedral and the Bazaar" (**www.tuxedo.org/~esr/writings/cathedral-bazaar/**) that make a case that open source development within an extended developer community results in better software. In this book, we discuss how a commercial company or an organization of any kind can build a business or extend an existing business through the creation and distribution of open source software—and why it's a good idea. In other words, we show you how to set up shop in the bazaar.

Potentially, moving to open source for a product can provide better value to your customers, including (in particular) allowing your customers or third parties to improve that product through bug fixes and product enhancements. In this way, you can create better and more reliable products that are likely to more truly reflect your customers' requirements.

However, the real benefit of building a business on open source software is to provide greater value to your customers than your competitors can, and ultimately to turn that increased value into increased revenue and profits for your company. In the traditional software business model, your company provides all (or almost all) of the value to customers, and you realize revenues and profits in return for that value through traditional software license fees. In an open source business model, you are not the only source of much of the value provided to customers; other developers who are attracted to working on your open source products will help augment your resources rather than your competitors'. These outside developers might be motivated by the prospect of working with software that solves important problems for them and for others and by the possibility of future gain in providing related services and creating related products. They might also be motivated by the opportunity to increase their own knowledge or by the ego satisfaction of building an enhanced reputation among their peers.

Thus, a significant part of your potential success depends on the work of others who work "for free"—that is, open source developers who contribute their work to your company and to the developer community at large without demanding or receiving any money or other tangible payment in return. However, open source developers will not (and should not) do this work unless you treat them fairly. This is in part a function of your company's attitudes and actions toward developers working with its products, but it is also formalized in the company's choice of an open source license, specifying the terms and conditions under which the company's open source products can be used, modified, and redistributed.

Open Source Licenses

There have been several standard license agreements published for use with open source software. All of them have some common features, most notably making software free to users both in terms of having no cost and in terms of minimizing restrictions on use and redistribution. These features are necessary for developers to feel fairly treated. If possible, you should use one of the existing open source licenses (see Appendixes A and B for examples) or modify one of those licenses to meet your needs; some licenses work better than others for particular business models. Possible license choices include:

- No license at all (that is, releasing software into the public domain)
- Licenses such as the BSD (Berkeley Software Distribution) License that place relatively few constraints on what a developer can do (including creating proprietary versions of open source products)
- The GNU General Public License (GPL) and variants that attempt to constrain developers from "hoarding" code—that is, making changes to open source products and not contributing those changes back to the developer community, but rather attempting to keep them proprietary for commercial purposes or other reasons
- The Artistic License, which modifies various of the more controversial aspects of the GPL

- The Mozilla Public License (MozPL) and variants (including the Netscape Public License or NPL) that go further than the BSD-like licenses in discouraging software hoarding, but allow developers to create proprietary add-ons if they wish

Open Source Business Models

Because you can't use traditional software licenses and license fees with open source software, you must find other ways of generating revenues and profits based on the value you are providing to customers. Doing this successfully requires selecting a suitable business model and executing it well. The following business models potentially are usable by companies creating or leveraging open source software products (see **www.opensource.org/advocacy/case_for_business.html** for examples and more information on the first four models):

- *Support Sellers*—Company revenue comes from media distribution, branding, training, consulting, custom development, and post-sales support instead of traditional software licensing fees
- *Loss Leader*—Company uses a no-charge open source product as a loss leader for traditional commercial software
- *Widget Frosting*—For companies that are in business primarily to sell hardware and that use the open source model for enabling software such as driver and interface code
- *Accessorizing*—For companies that distribute books, computer hardware, and other physical items associated with and supportive of open source software
- *Service Enabler*—Companies create and distribute open source primarily to support access to revenue-generating online services
- *Brand Licensing*—A company charges other companies for the right to use its brand names and trademarks in creating derivative products
- *Sell It, Free It*—Software products start out their product life cycle as traditional commercial products and are converted to open source products when appropriate
- *Software Franchising*—A combination of several of the other models (in particular Brand Licensing and Support Sellers) in which a company authorizes others to use its brand names and trademarks in creating associated organizations doing custom software development (in particular, geographic areas or vertical markets) and supplies franchises with training and related services in exchange for franchise fees of some sort

An organization might find any one of these business models—or a combination thereof—far more value building and ultimately more stimulating than the traditional approach.

How It All Started

The open source system did not somehow emerge spontaneously from the chaos of the Internet. Although you could argue that something like this system was bound to evolve eventually, the process was greatly accelerated by the stubbornness of one man: Richard Stallman.

Stallman's Idea

Informal code sharing had been around for a long time, but until Stallman gave the phenomenon a name and made a cause of it, the participants were generally not aware of the political consequences of their actions. In the 1970s, Stallman worked at the Massachusetts Institute of Technology's Artificial Intelligence Lab. The Lab was, in his own words (see his essay at **www.gnu.org/gnu/thegnuproject.html**), a "software-sharing community," an environment in which changes to program source code were shared as naturally as the air in the room. If you improved the system, you were expected to share your modifications with anyone else running a similar system so everyone could benefit. Indeed, the phrase "your modifications" is itself misleading; the work was "yours" in an associative sense, but not in a possessive sense. You lost nothing by sharing your work and often benefited further when someone else improved on your improvements.

This ideal community disintegrated around Stallman in about 1980. A computer company hired away many of the AI Lab programmers by paying them big money to do essentially the same work, but under an exclusive license. That company's business model was the same as that of most software shops today: Write a really good program (in this case, an operating system), keep the source code under lock and key so no one else can benefit from it, and charge a fee for each copy of the system in use. A "copy," of course, meant a binary copy. People outside the company could run the system, but they weren't allowed to see or modify the source code from which the executables were produced.

From the point of view of the former AI Lab programmers, the change might have appeared fairly minor. They still were able to share code with each other, because most of them went to work for the same company. However, they weren't able to share their code with anyone outside that company nor, for legal reasons, were they free to incorporate others' code into their products.

For Stallman, however, the prospect of hoarding code was intolerable; he'd had a taste of what a sharing community could be. Instead of accepting the supposedly inevitable and letting his community disappear, he decided to re-create it in a less vulnerable form. He started a nonprofit organization called the Free Software Foundation and began to implement a complete, free, Unix-compatible operating system, which he called GNU ("GNU's Not Unix"). Even more importantly, he designed a copyright license, the terms of which ensured the perpetual redistribution of his software's source code. Instead of trying to reserve exclusive copying rights to the author or owner of the code, the General Public License

(see Appendix A) prevented anyone from claiming exclusive rights to the work. If you had a copy of the work covered by the license, you were free to pass it around to others, but you could not require that others refrain from giving out copies. The rights had to be copied along with the code. These rights extended to modified versions of the work, so that once a work was covered by the GPL, no one could make a few changes and then resell it under a more restrictive license.

Stallman's idea caught on. Other people began releasing programs under the GPL and occasionally inventing similar licenses. In the meantime, the Internet was enabling programmers across the globe to have access to each other's code, if they chose to cooperate. Thus, the new software-sharing community came to include virtually anyone who wanted to join and had a Net connection, regardless of physical location.

At this point—about 1990—only a few people shared Stallman's confidence that public ownership of code was how all software ought to be. Even some regular contributors to the GNU project were not necessarily in favor of *all* software being free, pleased though they might have been with what the GNU project had accomplished so far. Before long, though, the movement (if it could be called that yet) received a tremendous psychological boost from the appearance of some completely free operating systems. In Finland, Linus Torvalds had reimplemented an entire Unix kernel (called *Linux*) and published his source code under the GPL. Combined with the Unix utilities already available from the GNU project, this became a usable distribution of Unix. Not long afterwards came the release of 386BSD, based on the BSD version of Unix, whose development had actually started before Linux. These were soon followed by the confusingly named NetBSD, FreeBSD, and, more recently, OpenBSD.

The appearance of entirely free operating systems was a real boon for the movement—and not just in technical terms. It proved that free code could result in quality software (in many situations, the free systems performed better and crashed less often than their commercial competitors). Because the vast majority of applications that ran on these systems were also free, there was a dramatic increase in the free software user base and, therefore, in the number of developers contributing their talents to free software.

The Two Types of Development

As more users removed commercial operating systems from their computers and installed free ones, the rest of the world (by which we mean nonprogrammers) began to notice that something unexpected was happening. With his usual timeliness, Eric Raymond published a paper, "The Cathedral and the Bazaar", which partly explained why free software was often so technically successful. The paper contrasted two styles of software development. The first, "cathedral-style," is tightly organized, centrally planned, and is essentially one creative act from start to finish. (Actually, we rather doubt that real cathedrals are built this way, but that's a topic for another time.) Most commercial software is written cathedral-style, with a guru heading up a team and deciding what features go into each release.

The other style resembles, in Raymond's memorable phrase, "a great babbling bazaar of differing agendas and approaches (aptly symbolized by the Linux archive sites, which would take submissions from *anyone*) out of which a coherent and stable system could seemingly emerge only by a succession of miracles." But emerge it did, and Raymond hit on the key reason for the recurrence of the miracle: "Given enough eyeballs, all bugs are shallow" (Linus's Law). The trouble with the cathedral style is that it fails to enlist the software's most natural ally—the users. A small (or even medium-sized) team of developers quickly becomes overwhelmed with the influx of bug reports and feature requests, and must spend a fair amount of time simply prioritizing and figuring out what to do next. Even after they know what they want to do, there's no telling how long it will take to track down a particular bug and come up with a maintainable solution. The result is that part of the development team spends too much time solving these issues and is unavailable for other work.

Furthermore, commercial development teams often operate under constraints (budgets, deadlines, and marketing strategies) unrelated to the technical problems of the software. Even the decision to continue maintaining a certain program is often based on business factors having little to do with the software's intrinsic quality and potential.

The users, on the other hand, just want good code. They want a useful program, they want the bugs fixed, and they want appropriate features added and inappropriate ones rejected. In retrospect, the solution seems obvious: Why not give the users the freedom to make all this happen themselves? Even though the vast majority of users are not programmers and cannot contribute to actually changing the code, those few who can will end up benefiting everyone.

What Does CVS Have to Do with It?

As with any popular movement that experiences sudden growth, the free software movement soon discovered that it had logistical problems. It was no longer enough for a free software author merely to place her code on a public Internet server and wait for people to download it. What if people downloaded it and then sent in hundreds of bug fixes and code contributions? For a popular program, no single author could be expected to organize and integrate all this feedback and still have time to write original code. In a closed source, centrally directed software company, the number of developers is relatively small and well paid, and the tasks are divided in advance. However, the open source author is often an unpaid volunteer who has no idea where the next useful snippet of code will come from or what that snippet will do. If she's lucky, she might have an organized core group of co-developers who can help fix bugs and review incoming contributions to ensure that they meet the project's standards. This type of group probably has a high turnover rate, though, because its members are likely also volunteers.

A geographically distributed volunteer organization obviously cannot afford to spend weeks or months training its members to work together, only to lose this investment whenever a member leaves the group and is replaced by a newcomer. A base set of conventions for

contributing to shared projects was also necessary—so newcomers could fit in easily—as well as an automated system for accepting contributions and keeping everyone up to date with changes to the code. These requirements are not, of course, unique to free software, but they were particularly pressing in this case because volunteers are less able to devote resources to management and are more likely to seek automated solutions that don't take a long time to learn.

diff and **patch**

The groundwork for such a system had already been laid. The standard Unix **diff** program knew how to reveal concisely the differences between two files. If you "take the diff" (as we say in the vernacular) between a file before a given modification and the same file afterward, the resulting diff—that is, the output of the **diff** program—consists of only the modification and omits those parts of the file that remain unchanged. A trained eye can look at a diff and know approximately what happened to the file; more importantly, a trained program can look at a diff and tell *exactly* what happened. Therefore, **diff** was soon augmented, to no one's surprise, by **patch**. Written by Larry Wall and released as free software, **patch** was to **diff** as integrals are to derivatives. If you take the difference between file A and file B (remember that B is often just A after some modifications) and feed it to **patch** along with one of the two files, **patch** can reconstruct the other file. (One result of this was that diffs soon came to be called "patches" and that's how we'll usually refer to them in the rest of this book.)

If this seems of dubious utility to you, put yourself in the position of the software developer who needs to accept code contributions from outside sources. A contribution, in practical terms, consists of a set of changes to various files in the project. The maintainer wants to know exactly what those changes are—what files were modified and how. Assuming the changes pass muster, he or she wants to put them into the code with a minimum of fuss. The ideal way to accomplish this is to receive a series of patches that can be inspected by eye and then automatically incorporated into the current sources via the **patch** program. (In real life, of course, the maintainer's sources might have other changes by that time, but **patch** is smart enough to perform fuzzy matching, so it usually does the right thing even if the files are no longer exactly the same as the ones used to produce the patch.)

With **diff** and **patch**, there was a convenient, standard way to submit contributions; however, programmers soon recognized a further need. Sometimes, a submission was incorporated into the sources and had to be removed later because it contained flaws. Of course, by that time, it was hard to figure out who applied what patch when. Even if programmers could track down the change, manually undoing the effect of a patch long after the fact is a tedious and error-prone process. The solution was a system for keeping track of a project's history—one that allowed the retrieval of previous versions for comparison with the present version. Again, this problem is not limited to free software projects—it is shared by the commercial world and various systems have been written to solve it. Most free software

projects, as well as quite a few commercial ones, chose Walter Tichy's Revision Control System (RCS), which is free and also relatively portable.

RCS

RCS did the job, but in hindsight, it lacked several important features. For one thing, it dealt with projects in a file-centric way; it had no concept that the various files of a project were related, even though they might all be in the same directory tree. It also used the "lock-modify-unlock" style of development, in which a developer wishing to work on a file first "locked" it so no one else could make changes to it, then did her work, and then unlocked the file. If you tried to lock a file already locked by someone else, you either had to wait until they were done or "steal" the lock. In effect, it was necessary to negotiate with other developers before working on the same files, even if you would be working in different areas of code (and, predictably, people sometimes forgot to unlock files when they were finished). Finally, RCS was not network-aware. Developers had to work on the same machine where RCS's per-file historical data was kept or resort to clumsy handwritten scripts to transfer data between their working machines and the RCS server.

The Winner: CVS

Thus was born the latest (and maybe not the last) in this progression of tools: CVS, or Concurrent Versions System. CVS addresses each of the aforementioned problems in RCS. In fact, it started out as a collection of scripts (written by Dick Grune in 1986) that were designed to make RCS a bit easier and were posted to the Usenet newsgroup **comp.sources.unix**. In 1989, Brian Berliner rewrote CVS in the C programming language, and Jeff Polk later added some key features.

CVS actually continued to use the original RCS format for storing historical data and initially even depended on the RCS utilities to parse that format, but it added some extra abilities. For one thing, CVS was directory-aware and had a mechanism for giving a group of directories a name by which they could be retrieved. This enabled CVS to treat a project as a single entity, which is how people think of projects. CVS also didn't require that files be locked and unlocked. Instead, developers could hack away at the code simultaneously and, one by one, register their changes into the repository (where the project's master sources and change history are kept). CVS took care of the mechanics of recording all these changes, merging simultaneous edits to the same file when necessary, and notifying developers of any conflicts.

Finally, in the early 1990s, Jim Kingdon (then at Cygnus Solutions, now at Cyclic Software) made CVS network-aware. Developers could now access a project's code from anywhere on the Internet. This opened code bases to anyone whose interest was sparked, and because CVS intelligently merged changes to the same files, developers rarely had to worry about the logistics of having multiple people working on the same set of sources. In a sense, CVS did for code what banks do for money: Most of us have been freed from worrying about the logistics of protecting our money, accessing it in faraway places, recording our major transactions, sorting

out concurrent accesses, or accidentally spending more than we have. The bank automatically takes care of the first four and notifies us with an alarm when we've done the last.

The CVS way of doing things—networked access to sources, simultaneous development, and intelligent automated merging of changes—has proven attractive to closed-source projects as well as free ones. At present, both worlds use it frequently; however, it has really become dominant among the free projects. A central thesis of this book is that CVS became the free software world's first choice for revision control because there's a close match (watch out; we almost said "synergy") between the way CVS encourages a project to be run and the way free projects actually do run. To see this, we need to look a bit more closely at the open source process.

Principles of Open Source Development and How CVS Helps

Programmers have long known that they could work together in physical and temporal separation from each other. The phenomenon has grown to the extent that it has its own academic/professional literature and acronym: CSCW (Computer Supported Collaborative Work). Although sites such as **www.sourceforge.net** are burgeoning with collaborative, open source software projects, little seems to be happening in the way of collaborative content development.

The principles Eric Raymond outlines in his essay "The Cathedral and the Bazaar," although aimed squarely at programmers, are perfectly valid for use in the development of content. The development model relies on and succeeds because of the interest and effort of talented authors and the truth of Linus's Law.

The first principle is that the source code is made accessible to the entire world (a major shift, if one is accustomed to proprietary software development). Instantly, a question arises: When should the source code be made available, and how often? At first glance, it would seem that the most recently released version would suffice, but if others are to find and fix bugs, they need access to the latest development sources, the same files the maintainers are working on. It's terribly discouraging to a potential contributor to spend days tracking down and fixing a bug, only to discover on submitting the patch that the bug has already been found and fixed. As any programmer knows, a release is just a snapshot of a development tree at a particular moment. It might be an unusually well-tested snapshot, but from the code's point of view, the released version is not qualitatively different from a snapshot taken at any other time. As far as contributing authors are concerned, a free software project is in a state of continuous release.

Unfortunately, traditional methods of software distribution weren't designed for continuous, incremental updates. They were designed around the idea that a release is a monumental event, deserving special treatment. In this "grand event" way of doing things, the release is

packaged into a static collection of files, detached from the project's past history and future changes, and distributed to users, who stay with that release until the next one is ready, sometimes months or years later. Naturally, the development sources do not remain static during that time. All of the changes that are to go into the next release start slowly accumulating in the developer's copy of the sources, so that by the time the new release date nears, the code is already in a substantially different state from the previous release. Thus, even if full source code were included in every release, it still wouldn't help much. Users soon would be working with out-of-date files and have no convenient way to check the state of the master sources accessed by the maintainer and core developers.

For a while, this situation was handled with workarounds—partial solutions that were not terribly convenient but could at least be tolerated. Snapshots of the development sources were made available online on a regular basis, and any users who wanted to keep up with the project's state could retrieve those sources and install them. For those who did this regularly, the process could be partially automated by scripts that retrieved and installed each "development release" nightly. However, this is still an unsatisfactory way to receive changes. If even one line of code in one file changed and everything else stayed the same, the interim release would still have to be retrieved in its entirety.

The answer (you knew this was coming) is CVS. In addition to giving active developers a convenient way to enter their changes into the master repository, CVS also supports anonymous, read-only access to the repository. This means that anyone can keep a development tree on his or her local machine, and when the programmer wants to start working on a particular area of code, he or she simply runs one command to make sure the tree is up to date. Then, after checking to make sure that the problem hasn't already been fixed in the batch of changes just received, the programmer begins to work. Finally, when the changes are ready, CVS automates the process of producing a patch, which is then sent to the maintainers for inspection and possible incorporation into the master source tree.

The point here is not that CVS makes something possible that previously was impossible; retrieving up-to-date sources and producing patches were all theoretically possible before CVS appeared. The difference is that CVS makes it *convenient*. In a system that relies largely on volunteer energy, convenience is not a mere luxury—it is often the factor that determines whether people will contribute to your project or turn their attention to something with fewer obstacles to participation. Projects are competing for volunteer attention on their merits, and those merits include not only the quality of the software itself, but also potential developers' ease of access to the source and the readiness of the maintainers to accept good contributions. CVS's great advantage is that it reduces the overhead involved in running a volunteer-friendly project by giving the general public easy access to the sources and by offering features designed specifically to aid the generation of patches to the sources.

The number of free software projects that keep their master sources in CVS is impressive by itself. Even more impressive is that some of those projects are among the largest (in terms of number of contributors) and most successful (in terms of installed base) on the Internet.

They include the Apache WWW server, the GNOME free desktop environment, FreeBSD, NetBSD, OpenBSD, the PostgreSQL database, the XEmacs text editor, and many more. In later chapters, we'll examine in detail how projects use CVS to manage their sources and aid their volunteers.

What Makes It All Tick?

Until now, we've focused on the advantages of free software for users. However, developers still face an interesting choice when they consider free software. As long as copyright law exists in its current form, it will probably always be more lucrative for a programmer to work on proprietary code—the profits can be enormous (even after illegal sharing is taken into account) when each running copy of a popular program is paid for individually. If you want to get rich, your course is clear: Write a useful piece of closed-source software, get it noticed, and wait for Microsoft to make an offer for your company.

Yet somehow, free software projects still manage to find programmers. There are probably as many different explanations for this as there are people writing free code. Nevertheless, if you spend enough time watching mailing lists and developer discussion groups, a few core reasons become apparent: necessity, community, glory, and money—not necessarily in that order and certainly not mutually exclusive.

Necessity

Eric Raymond hypothesizes that the first reason, necessity (the need to "scratch an itch"), is the chief reason why most free software projects get started at all. If you just want a problem solved, once and forever, and you aren't looking to bring in any revenue from the code (aside from the time you'll save by using it), then it makes a lot of sense to release your program under a free license. If you're lucky, your solution will turn out to be useful to other people, and they'll help you maintain it. Call it the Kropotkin Factor—sometimes, cooperation is simply the most winning strategy.

Community

Our favorite reason, though, is actually the second: community. The sheer pleasure of working in partnership with a group of committed developers is a strong motivation in itself. The fact that little or no money is involved merely attests to the strength of the group's desire to make the program work, and the presence of collaborators also confirms that the work is valuable outside one's own narrow situation. The educational value of working with a group of experienced programmers should not be discounted, either. We've certainly learned more about programming from reading freely available code, following online discussions about the code, and asking questions, than from any book or classroom. Many active developers of free software would probably say the same thing. Most seem quite conscious that they are participating in a kind of informal, peer-to-peer university and will happily explain things

to a newcomer, as long as they feel the newcomer shows promise of contributing to the code base eventually.

Glory

Meanwhile, in the back of everyone's mind (well, not *yours* or *ours*, of course!), is glory—the fame that comes from occupying a prominent position on the developer team of a widely used free program. Most programmers with even a peripheral involvement in free software are likely to recognize the names Linus Torvalds and Alan Cox (for work on the Linux kernel), Brian Behlendorf (of the Apache Web Server team), and Larry Wall (inventor of, among other things, the popular Perl programming language). Raw self-aggrandizement might not be the most attractive motive, but it can be powerful and if properly harnessed, it can bring about a lot of useful code. Happily, in the free software culture, you can achieve glory only by sharing the benefits of your work rather than hiding them. Note that there is often no official (that is, legal) recognition of what constitutes a "prominent position" in a group of developers. People acquire influence by writing good code, finding and fixing bugs, and consistently contributing constructively in public forums. Such an unregulated system might seem open to exploitation but, in practice, attempts to steal credit don't succeed—too many people are too close to the code to be fooled by any false claims. A developer's influence in the community is directly proportional to the frequency and usefulness of her contributions, and usually everyone involved knows this.

One side effect is an uncommon scrupulousness about giving credit where credit is due. You've probably noticed that we're being careful to mention developers' names when talking about specific pieces of software. Giving credit by name is a common practice in the free software world, and it makes sense. Because the work is often done for little or no pay, the possibility that contributions will be recognized and reputations correspondingly enhanced makes the work attractive. Fortunately, another side effect of using CVS (or any version control system) is that the precise extent of every developer's modifications is recorded in the change history, which can be retrieved and examined by anyone at any time.

Money

Finally, there is money. People have begun to find ways to get paid to work on free software. In many cases, the wages are considerably more than a bare living, and even if not quite as lucrative as, say, stock options at a proprietary software company, the pleasure of seeing one's code widely distributed is often enough to compensate for a little income foregone.

One way for people to make money is to sell services centered on a particular code base. The software might be free, but expertise is still in limited supply. A common strategy is to specialize in knowing everything there is to know about a particular free tool or suite of tools and offer technical support, customizations, and training. Often, the company also contributes to the maintenance of the program (and no wonder, because it's in the company's interest to ensure that the code remains healthy and free of bitrot). More recently, companies have

begun to specialize in packaging particular distributions of free software and trading on the "brand name" they earn through making reliable bundles. Oddly enough, this actually seems to work. Red Hat Software has been profitably selling Linux distributions on CD-ROM for several years, despite the fact that anyone is free to copy its CDs and resell them or even download the software directly from Red Hat. Apparently, the reliability of its distribution is important enough to consumers that people will pay a little more for the extra reassurance of getting it on CD-ROM directly from Red Hat instead of from a reseller.

Also, hardware companies sometimes devote resources to guaranteeing that popular free applications run on their machines. If a company formerly offered proprietary software as a sideline to its hardware and service businesses, it might now ship free software that it has tested and perhaps modified to perform better on its hardware. You might think that the company would want to keep its modifications secret (were that permitted by the software's license in the first place), but it turns out to be entirely to the company's advantage to release any changes back into common distribution. By doing so, the company avoids having to shoulder the entire maintenance burden itself. By releasing the source, it has empowered its users to give feedback on whether the program runs well on the hardware. (The goodwill thus gained among its customers might also be a factor.) Because it isn't in the software business anyway, the hardware company is not looking for a direct return on investment in that area.

The arrival of big money into the formerly pure free software world has not been seen universally as a positive development and, in fact, has led to some rather heated debate about the ultimate purpose of free software. To attempt to summarize that debate is like diving into shark-infested waters, indeed; however, it's a significant issue right now, so we'll don shark repellent and do our best. The issue arose because free software has been so technically successful. Stable, bug-free software—whatever its origins—is something any business wants to offer its clients, as long as doing so doesn't conflict with any other goals (such as increasing sales of one's own closed-source software). In the for-profit consulting world, the innate quality of the software, in a purely technical sense, is the only concern. If the best product is free software, so be it; if it's something else, use that instead.

The terms of redistribution are not a major factor in the decision to use the software, except in how that affects one's ability to use it. Thus, some people are for free software simply because it leads to better code, which can also imply reduced hassle and higher profits. For others, however, cooperation itself is the goal. Richard Stallman is one of the most forceful evangelists for this position (for him, sharing information is a moral crusade), but he is not alone in viewing profit-driven development with distrust.

Factionalism as a Sign of Strength

Although we personally lean toward the "cooperation is the goal" attitude, we also don't think free software is really threatened by the influx of corporate money. For free software, the only truly important currency is developer attention. To the degree that corporate money

subsidizes developers who devote time to free software, it helps the software and the community. When that money is used to pay for closed-source software, programmers will still create and maintain free code, and that code will continue to be of high quality. That's simply what many programmers want, and what they do on their own time is up to them. Perhaps occasionally, a company will promote a program as "open source" when it's not and briefly tempt a few developers into wasting their time with code that is not free. However, the legal language of the software's license is open for inspection, and no amount of marketing or propaganda can make it mean something it doesn't. Inevitably, developers realize this and turn their attention to truly free work.

In the end, the appearance of factionalism (of which the disagreement about the role of money is only one example) in the free software movement is probably a sign of strength. It means that people are now secure enough about free software's success that they no longer feel the need to present a unified public front or avoid rocking the boat. From here, it's merely a matter of taking over the world.