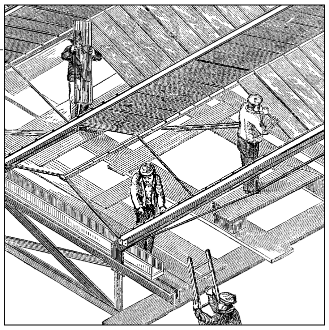
CHAPTER 7

Software Development Using Open Source and Free Software Licenses



The purpose of open source and free software licensing is to permit and encourage the involvement by licensees in improvement, modification, and distribution of the licensed work. This open development model of software development is the unique strength of the open source and free software movement. While the open source and free software licenses already discussed approach open software development differently, open development is the goal.

This chapter describes the basic principles of software development under open source and free software licenses, including the problems of forking, community development under the bazaar and the cathedral models, how open source and free software projects are initiated and maintained, and the effect that license choices can have on software development. This chapter also briefly discusses the basic principles of drafting contracts, for those who are interested in drafting their own software license.

Models of Open Source and Free Software Development

The open source and free software licensing is driven by the development model, or models, that it is intended to encourage. After all, there is little point to permitting the "free" modification and distribution of a work if people do not actually take the opportunity to modify and distribute the licensed work.

These licenses are intended to permit, and indeed, to encourage the contributions of others to the project. Nonetheless, one of the first open development projects relied, at least at the beginning, on a relatively small number of closely-knit developers. This project was Richard Stallman's plan to develop a complete operating system modeled after the Unix operating system but written entirely in free code.[*]

[*] The following discussion draws heavily from the essay of Eric Raymond, "The Cathedral and the Bazaar," in *The Cathedral & The Bazaar: Musing on Linux and Open Source by an Accidental Revolutionary*, Eric S. Raymond (O'Reilly, 2001).

This project created numerous, deeply influential programs, including the widely used Emacs and the GNU C Compiler and, with the arrival of the Linux kernel developed by Linus Torvalds and his associates, resulted in the creation of the first entirely free operating system, the GNU/Linux operating system. Stallman is also the author of the GPL, and the first, and still most important, philosopher of the free software movement.

Nonetheless, the initial projects under the aegis of the Free Software Foundation—the group Stallman founded to serve as the homebase for the nascent free software movement—did not rely on the open development model, to the same extent, for example, as the Linux project did. Part of the explanation for this is purely a matter of circumstance. The great engine of free software development is the Internet. When Stallman had his epiphany as to the importance of keeping software free in the early 1980s, the Internet was still in its early adolescence. While universities and colleges (particularly those associated with the Department of Defense) and scientific institutions had access to it, relatively few individuals did.

Stallman originally announced his intention to create a complete Unix-compatible software system in the fall of 1983. At that time, he had already written the widely popular Emacs editor, and he started to develop a completely free operating system. The frustration that Stallman felt with the increasing strictures placed on free computing and in particular with the application of security protocols, passwords, and "blackbox" binary code that drove him to this project has been well-described elsewhere.* After he formally resigned from the Massachusetts Institute of Technology's Artificial Intelligence lab, Stallman dedicated himself to creating various components that would become critical parts of the GNU/Linux operating system: the GNU C Compiler, GNU Emacs, the GNU Debugger, the GNU C Library, and perhaps no less importantly, the GNU Public License.

It is no exaggeration that it was Stallman's original intention, and his practice for a considerable period, to undertake the bulk of the work substantially by himself. An episode from around the time of the beginning of the GNU project demonstrated that this was possible. By 1982, a company named Symbolics had hired away more than a dozen programmers from the MIT AI Lab to develop a commercial version of the Lisp operating system—an operating system developed and maintained by the MIT AI Lab— against a competing company, Lisp Machines, Inc., or LMI, which had also hired numerous MIT hackers. Under its agreement with MIT, Symbolics was contractually required to permit Stallman, as MIT's administrator of the Lisp system, to review the source code but not required to permit MIT to adopt any of that code. Nonetheless, Symbolics, as a matter of custom, permitted Stallman to adopt features from its source code and maintain them in MIT's version of Lisp. Stallman kept MIT's version of Lisp free, and LMI looked to it to see what developments and improvements its competitor, Symbolics, had made.

* The circumstances surrounding Stallman's decision to begin work on the GNU project are described in *Free As In Freedom: Richard Stallman's Crusade for Free Software*, Sam Williams (O'Reilly, 2002).

In early 1982, Symbolics decided to hold MIT to the terms of the agreement and barred Stallman from incorporating changes from its version of Lisp. Stallman viewed this as a declaration of war. In what is still considered one of the major feats in programming history, Stallman spent much of the next two years matching the new features and additions in Symbolics' Lisp on his own, keeping pace with a much larger team of programmers, feature for feature.

In the period from early 1984 to 1990, Stallman was generating useful and influential programs at a phenomenal rate. In addition to the GNU Emacs, the GNU Debugger, and the GNU C Compiler already mentioned, Stallman developed GNU versions of several Unix programs, including the Bourne shell, YACC, and awk programs. However, in developing these programs, Stallman relied heavily on his own immense facility as a programmer and a relatively small number of collaborators. While the GPL was designed to ensure maximum freedom to users and programmers for programs developed under the license, Stallman himself, as a project manager, maintained relatively tight supervision over each of the GNU projects.

This led, perhaps inevitably, to the first major stumbling block of the GNU project. Stallman, quite deliberately, had organized his operating system around a piecemeal approach in which the tools for the system would be written before the kernel, its central component. By 1990 or so, that kernel was the last major piece not to have been completed. Stallman and the GNU project had been working on a kernel since at least 1987, starting first with a kernel based on Trix, an MIT program. By 1993, however, the GNU project, having abandoned Trix, had gotten bogged down in a micro-kernel called Hurd.

There were a number of issues that slowed the development of Hurd, including the focus by a more mature Free Software Foundation on the theoretical aspects of micro-kernel development; a breakdown in communication between the GNU Debugger group and the group in charge of developing the kernel; "look and feel" lawsuits that had been brought by Apple and Lotus against other operating systems (most notably Microsoft); and perhaps not least, limitations on Stallman's own contibutions, caused by a disability that prevented him from typing.* This temporary setback set the stage for another great open development project, one using a very different development model.

Just two years earlier, in 1991, Linus Torvalds had started work on his own operating system kernel. Originally based on the Minix operating system, itself an "open" operating system designed for teaching purposes, in a famous email on August 25, 1991, posted to the Minix usegroup, Torvalds announced that he was working on a "(free) operating system (just a hobby, won't be big and professional like gnu) for

* For a more detailed discussion of the Hurd micro-kernel and the difficulties in its development, see *Free As In Freedom: Richard Stallman's Crusade for Free Software*, Sam Williams (O'Reilly, 2002) at pages 146 and following.

386 (486) AT clones."[*] By September, Torvalds had released the first version of Linux, Version 0.1. Interest in Torvalds' operating system, at least within the relatively small Minix community, was immediate and intense. Other programmers quickly responded to Torvalds' postings with questions, comments, and suggestions for how to improve the nascent operating system.

These postings set into motion what would quickly become the Linux phenomenon. This process involved, and indeed depended on the contributions of at first dozens, then hundreds, and now thousands of users, debuggers, and programmers. This development model is likely Torvalds' most significant contribution to open source and free software programming—notwithstanding his own considerable organizational and programming abilities. As the project grew in size and complexity, a structure developed organically, with other noteworthy programmers—such as Alan Cox, Dave Miller, and Ted Ts'o—taking on significant roles in managing the burgeoning growth of these projects. These three, and others, act as intermediaries between Torvalds, who remains at the center of the project.

As Eric Raymond put it in his essay "The Cathedral and The Bazaar," "Linus's cleverest hack was not the construction of the Linux kernel itself, but rather his invention of the Linux development model."[†] As described by Raymond, this development model is dependent on a number of interlocking conditions. The first is the importance of users. Every program needs a constituency of users who use the program, want the program to work, and are sufficiently committed to make at least some effort toward improving it, whether it be by contributing bug reports or patches. The consistent involvement of such users makes the discovery and elimination of bugs easier. The second is the maxim of "release early, release often." By releasing early and quickly incorporating changes from users, project developers keep their user base actively engaged and involved. When a user notices a bug, submits a patch, and then a few weeks (or even days) later sees the improvement he suggested worked into a new release, he sees immediately the benefits of the development model. He has been rewarded, not financially, but by the availability of a better program. This reward, of course, is shared within the entire community of developers. The "release early, release often" strategy also cuts down on the possible duplication of effort by a number of users/programmers working, unknown to each other, to identify and fix the same bug. When a problem is quickly identified and its solution is incorporated into a new release, the number of users (and hence potential debuggers) exposed to that solved problem is reduced.

This debugging strategy takes advantage of the many different perspectives, and different uses, put to the program by a spectrum of users. While a bug may seem difficult to isolate from the perspective of a single programmer, that same bug may, upon

[*] Torvalds' email as reprinted in *rebel code: inside linux and the open source revolution*, Glyn Moody (Perseus Publishing, 2001) at page 42.

[†] *The Cathedral & The Bazaar: Musing on Linux and Open Source by an Accidental Revolutionary*, Eric S. Raymond (O'Reilly revised ed. 2001) at page 29.

exposure to a hundred different users and programmers, seem immediately obvious to just one of them. As long as that one is sufficiently committed to submit a detailed bug report or a patch, the project has progressed, and probably more quickly and easily than a more tightly focused, but smaller, group of programmers would have reacted.

This debugging perspective does not necessarily address the complex problems of organizing group work on developing source code in the first instance. In such cases, depending on the development model, adding more programmers to a project may not quicken development, but in fact may slow it down as the additional costs associated with communicating information among a larger group of people outweigh the incremental benefit of adding programmers to a project. While the Linux development model has kept direction and focus within a relatively small circle, as may well be necessary for a software project of any size to survive, much less one of the size and complexity of Linux, its openness has been its strength. By encouraging "egoless" contributions that are improvements to an already established workflow, as opposed to redirections of that workflow, the Linux development model avoids much of the drag that can result from the difficulties in social and information engineering in large, traditional, software projects.

This bazaar model contrasts with what Raymond describes as the cathedral model of software development. Software development, in its traditional form, relies on tightly focused, relatively small groups of programmers associated with a single institution or corporation. Such groups sometimes are as small as two or even just one programmer. Unix itself was the creation of legendary hacker Ken Thompson at Bell Labs: it was written in the programming language C, itself written by another hacker, Dennis Ritchie. Both Unix and C were designed to be simple (or at least simpler than their contemporary competitors). This simplicity and their immense popularity made them prototypes for Linux and the GNU programs that came after them.

Their simplicity and portability made them popular among programmers. Despite an almost total lack of interest by AT&T (Bell Labs' corporate parent), Unix and C spread quickly, first inside AT&T and then outside it. By 1980, it was commonplace in universities and research institutions. Unix, the model for the GNU project and Torvalds' Linux project, set the stage for open source development.

Nonetheless, Unix itself never became a truly open development.* Although there were a number of "hot-spot" programming communities—including Berkeley, the AI and LCS labs at MIT, and Bell Labs itself—these communities were largely self-contained, and although relatively large in the number of programmers they had, did not have the mass to support an open development project, even if there was one. The

* It is an irony worth noting that the current holder of the rights to Unix, the SCO Group, has sponsored numerous continuing lawsuits against users of GNU/Linux distributions under the theory that some, as of this writing unspecified, portion of these distributions contains Unix code under the copyright held by SCO Group.

absence of such a project was in part due to the legally imposed limitations by trade secrets and copyrights, and movement toward commercialization of software in the late 1970s and early 1980s. The same trends that led to Stallman's Symbolics war and his subsequent exit from the MIT AI Lab were closing doors to open development projects. Software, once given away for free with expensive hardware, was becoming a booming business in itself.

In its traditional form, commercial software development is based on the exploitation of the monopoly created by copyright for competitive advantage. It makes sense in that system to avoid any process that would undermine that advantage, such as, for example, the sharing of source code with thousands of potentially competing strangers. Programmers for commercial concerns do "work-for-hire": the code they write does not belong to them but to their employers. They are routinely required to sign non-disclosure agreements, preventing them from disclosing to anyone else information that is proprietary (i.e., what their employer considers to be proprietary). Such programmers are also frequently asked to sign non-compete agreements, which prevent them from working for their employer's competitors for a year or two (or more) after they leave that employer. In this environment of deliberate concealment of any information that could be of use to the competition, the idea of open source is anathema.

This emphasis on secrecy channeled commercial programmers into cathedral-style models of software development. While such companies are free to hire as many programmers as they may need, even the resources of a company such as Microsoft are limited.* No user base (or almost no user base) would be willing to subject itself to the disclosure restrictions that are required to maintain the commercial advantage software companies want.† Without open source code and knowledgeable (and energized) users, bug reports, to the extent they are submitted, greatly diminish in value to the project. What results is a relatively small group of programmers, as talented as the resources and attractiveness of the company can gather, building the software project essentially in secret and presenting it as a black box to the software-buying public.

This model of software development is not limited to commercial development. The GNU project, while certainly not anywhere near as "closed" as traditional commercial software development, relied heavily on the contributions of a relatively small number of people who were relatively tightly organized. The GNU project did not, at least in its early days, follow a "release early, release often" model. Its ability (or desire) to incorporate bug reports and patches submitted by users outside the project was limited accordingly. This should not be read as a slight to the GNU project.

* Microsoft's Shared Source Initiative, briefly described in Chapter 5, is driven in large part by its attempt to engage with this problem, that is to say, to involve as large a group of developers and users in its process without surrendering its legal rights under copyright law.

† The Sun Community Source License, described in Chapter 5, with its restrictions on distributions outside the community of developers, is a step in that direction.

GNU Emacs has incorporated the suggestions of hundreds of participants over more than 15 years of development and stands as a highly respected model of free software development. In addition, the GPL built a foundation for the open development model.

What really accelerated the full bloom of the Linux development model, however, and the astonishingly rapid development of Linux itself, was what Raymond calls "cheap Internet." While the predecessor of the Internet, ARPANet had been available at most research universities and institutions since the 1970s, the available bandwidth was small and access was limited. The cascading expansion of the Internet from 1990 or so on allowed a whole new realm of users to access it for email, Usenet groups, and surfing the newly developed World Wide Web.

The availability of software archives accessible by the Internet, Usenet groups open to contributors, and most importantly, email to permit communication between project originators, contributors, and users, were all necessary for the success of the Linux development model on the scale that Linux itself has achieved. The legal infrastructure of open source combined with the technical infrastructure of the Internet to make this new approach possible.

The Linux development model is obviously not the only one for developing software. It depends on the commitment and knowledge of its user base to succeed. Such users simply may not be available for every type of program. End user applications (such as video games) have been slow to develop under open source or free software development models.

Nonetheless, the Linux development model is useful (and powerful in its applications) for much more than just Linux itself. The same Linux-style development has been used successfully for a large number of programs.

While the choice of a particular license is an important factor, it is far from the only factor in determining the development of any given project. Both Linux and the GNU project's many developments were created under the same license, the GPL.*
Nonetheless, as just described, they follow very different patterns of development. The circumstances surrounding the development of a project, and, in particular, the personalities of those involved and the technology available to its originators, developers, and users, can have far more to do with the success of a project than the choice of a particular license.

The open development model may even keep code "open" that the governing license would permit to be closed, by incorporating it into a proprietary license. For example, as described in Chapter 2, the Apache License permits distribution of modified versions under proprietary licenses. In June of 1998, IBM announced that it would ship Apache as part of its WebSphere group of programs and provide continuing

* The very first releases of Linux were released under an open source license of Torvalds' own devising. Torvalds, however, adopted the GPL early on and it has covered every subsequent distribution of Linux.

enterprise level support for it.[*] As a natural consequence of this adoption, IBM developed its own modifications to the Apache software and distributed them under a license that it had written for this purpose, the IBM Public License. The original Apache license permitted IBM to license its modifications under a proprietary license and not to disclose their source code, and the IBM Public License did nothing to limit its ability to do so. Nonetheless, IBM continued to publish its source code and to freely permit the adoption or modification of its own work. The reason for this was simple. If IBM kept its code proprietary, eventually its version of Apache would depart from the standard Apache version. Future modifications to the standard version would become more difficult to port to IBM's version. IBM would lose the benefits of the open development process for its own version of Apache, as users and potential contributors would have less incentive to contribute bug reports or patches to it—particularly when a strong competitor, such as standard Apache, existed in the same marketplace.

In short, if IBM wanted to remain a contributor to the process (as well as a beneficiary in the fullest sense), it had to contribute, or at least not to keep whatever contributions it had already made to itself. Regardless of the terms of either of the applicable licenses, IBM's or Apache's, to get the full benefits of open source development, IBM had to live by open development rules.

Forking

By maintaining its own Apache development as an open development project, IBM avoided creating a fork. Forking occurs whenever a software project splits. While the two versions may remain entirely or partially compatible for some period of time, inevitably the unique (and now distinct) histories of each one's development will push them apart.

Forks can happen for many different reasons and may have entirely healthy consequences. A very simple piece of code may be developed by a group of programmers to do, for example, packet-switching. One half of the group may decide to follow a development tree leading toward making the simple packet-switching program into a complex database, and the other half may want to make the same program into a video-on-demand server for use in cable television systems. Such forks can occur without rancor and without any real concern for duplicative or unnecessary programming; the two future developments are so starkly different that mutual compatibility is of no concern.

Forks in more mature projects, however, are much more capable of producing undesirable results. For example, in 1993, the GNU Emacs project forked. Jamie Zawinski led a group of other developers on a line separate from that of Emacs' creator

[*] The circumstances surrounding IBM's decision to support Apache are described in *rebel code: inside linux and the open source revolution*, by Glyn Moody, (Perseus Publishing, 2001) at page 205 and following.

Richard Stallman and the GNU project. In part, this fork was driven by real differences as to the best course of future development for Emacs, but it also may have been the result of personality conflicts and concerns with the progress of Emacs development. Some felt that Stallman was relying too much on his own efforts and those of other programmers from the GNU project, thereby slowing development of Emacs. The fork was successful in the sense that two Emacs development projects resulted; as of this writing, both projects are continuing with no indication that this fork will ever close.

Forks in mature projects are properly feared. In addition to creating hard feelings, such forks undermine the foundation of the open development process. They split the user base as well as the programmers that contribute to the project. Given the importance of users to open development, this is a result to be avoided. While two open development projects may remain sufficiently similar for some period of time that modifications and bug patches can be ported from one project to another, at some point, the developments will have diverged sufficiently such that porting a solution from a competing project is no easier than developing that same solution from scratch. This duplication of effort and division of the development community for what, after all, are likely to be two very similar programs, argues strongly against such a fork, except under exceptional circumstances.

Given the serious consequences of forking, it is not unreasonable to look to licenses to prevent or at least to decrease the probability of such forks. While no open source or free software license is fork-proof, they do provide varying levels of protection against such forks.[*] Some licenses, such as the Apache and Perl Licenses, rely largely on the reputation of the project developers to avoid forks, but they also include some terms that shore up that defense against forks. Other licenses, such as the GPL, at least hinder forking by requiring that developers distribute or modify the licensed code only under "open development" terms. However, by permitting non-open development of code developed under them, code licensed under the MIT or the BSD License may be more prone to forking than code licensed under other licenses.

The network security program Kerberos was released under a variation of the X license that operates substantially like the MIT and BSD licenses. As described in Chapter 2, Microsoft adopted Kerberos and implemented it in its Windows 2000 (and subsequent) operating systems in a version that contained proprietary extensions for communicating with Microsoft servers. This was a fork in that because of these extensions, Microsoft's version of Kerberos is on a separate development plane than the MIT-distributed version of Kerberos and will likely continue to develop more proprietary extensions as Microsoft expands it. This was the result of the use of the X license, which has no terms that would prevent this development.

[*] Proprietary licenses are unforkable. There is no development by anyone other than the licensor and accordingly no possible foundation for a fork. Licenses such as the Sun Community Source License, while not open source licenses, head off forking by designating an official version by compliance testing and by prohibiting the commercial distribution of other versions.

As described in Chapter 2, the Apache License does not prevent the incorporation of its code into code licensed under another license, including a proprietary license. The Apache license does, however, include provisions protecting the Apache name. Specifically, the Apache license, Version 1.1 (as well as Version 2.0), prevents the use of the name Apache in connection with the work being distributed without permission, through Sections 4 and 5.

> 4. The names "Apache" and "Apache Software Foundation" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact apache@apache.org.
>
> 5. Products derived from this software may not be called "Apache" nor may "Apache" appear in their name, without prior written permission of the Apache Software Foundation.

Through this relatively simple device, maintaining a monopoly over the name—if not the licensed code—and maintaining a dynamic high quality distribution, the Apache Software Foundation has remained as the center of Apache development and avoided any forks of consequence.

A similar strategy works in the Artistic License that applies to Perl. As described in Chapter 4, the Artistic License defines both a Copyright Holder and a Standard Version of the program. Contributors to a program so licensed must either permit their modifications to be incorporated into the Standard Version, abstain from public distributions of their version of the work, or clearly document the changes in their version. While forking is possible under this license, the likelihood that any such fork would create a major competitor to the Standard Version is substantially reduced. Indeed, these provisions of the license—along with the steadfast commitment to Perl of its creator, Larry Wall, and his reputation in open source and free software development—have prevented any significant forks from developing in Perl to date.

The GNU GPL limits the likelihood of forks by prohibiting non-open development models for projects that incorporate GPL-licensed code. Every development project under the GPL can accordingly draw freely from every other project. After a fork of a GPL project, each leg of the project remains free to draw on the work of the other—to the extent such work may be available*—a process that may hasten the closing of such a fork and permit the reunification of the forked project. This is obviously not foolproof, as seen in the example of GNU Emacs.

Accordingly, while the choice of license certainly can have some effect on preventing forks, the nature of the open development model is conducive to forking. Permitting open access to source code and encouraging development by outsiders both allows for and creates incentives for the development of forks. Addressing forks is less a question of adopting the proper license, as any open source or free software license permits forking in some way, and is more a question of project development.

* While the GPL requires that code derived from GPL-licensed code also be distributed under the GPL, a developer can avoid "sharing" the code she has developed by simply not distributing it.

Choosing an Open Source or Free Software License

Choosing an open source or free software license is more often the result of circumstances than the unfettered discretion of a particular programmer. While each of the licenses described in this book (which represent only a selection of the open source and free software licenses in use) presents its own advantages and disadvantages, in many situations, the decision as to which license to apply will already have been made.

A typical route to involvement in an open source or free software project comes from contributing to an already existing project. Whether by submitting a patch to Linux or a bug report to a less well-established open development project, consideration of the license applicable to the project is generally a secondary consideration at most. A user submitting a bug report does not generally care about the license of the program to which the bug report applies. So long as that user can reasonably expect some benefit from the submission of the bug report, usually in the form of an improved program, that user will make a submission.

Users frequently make even more substantial contributions to open development projects without much more consideration. Again using Linux as an example, scores of programmers have submitted and continued to submit patches or more substantial contributions to Linux without troubling themselves to any great extent about the terms of the GPL applicable to Linux.

A programmer may undertake even more substantial responsibilities for an open development project by helping to maintain it or even taking a leadership role, without choosing the license applicable to the development. In the world of open source and free software, projects are frequently handed down, and the "successor" lead programmer takes over a project from the project's initiator. In such situations, the project comes with the license under which it was originally written. While a successor project leader could in theory insist that a new license apply to the project, the administrative and legal difficulties would have to weigh against such a switch. Even if the original project leader were agreeable, the new project leader would most likely need to secure the consent of every programmer who had contributed to the project under the previous license. After all, they had made their contributions with the understanding (to the extent that they had one) that what they contributed would be licensed under the license originally applicable to the project. Depending on the number of contributors, this could be a considerable hurdle.

Even for "new" open source or free software projects, the choice of a license may substantially be determined by license choices made by others. After all, given the nature of the open development model, it is frequently unnecessary to create a new program from scratch. Whatever the program's function, it is likely that someone, somewhere, has done something similar. By scanning SourceForge.net or other similar sites, someone considering an open source project can see whether a sufficiently similar project is already underway. Such a search might turn up an already existing open source project so similar to the one being considered as to make a new project unnecessary. In any event, prior work on similar projects in many situations will provide a foundation for a new project. In such a case, the developer has to consider carefully the license applicable to the pre-existing project.

Depending on the license, the developer may or may not have the ability to choose a different license to apply to the new project. If that pre-existing project uses an MIT or BSD-type license, the developer can use virtually any license, so long as the proper notification and disclaimer provisions are included. On the other hand, if the pre-existing project were licensed under the GPL, the developer would have little choice but to license his or her own project under the GPL or to get permission from the author to use a different license. As discussed in the previous chapter, different licenses provide different levels of compatibility with other licenses. Given a potential conflict between the provisions of two different licenses, it is the better practice to avoid the conflict entirely, either by developing the project under the same license as the pre-existing project, or by obtaining explicit permission, if possible, from the creator(s) of the pre-existing project to cross-license that project under the license to be used in the new project.

Accordingly, in many situations, a developer's choice of license is constrained by choices made by his predecessors. In fact, this is the intended purpose—described as having a "viral" effect on licensing decisions—of one of the most popular of the open source and free software licenses, the GPL.

In those situations in which a developer is in fact starting from scratch or from code whose license is amenable to change, the decision as to license will probably be largely a matter of personal preference. The factors that might influence this decision include: how frequently used and well-known the license is; how readily comprehensible that license is; and finally, and perhaps most importantly, the license's philosophy, and, in particular, the extent to which the license allows with code developed under other licenses, including proprietary licenses.

In choosing any license to apply to a new project, developers should strongly consider relying on those licenses already well-known in the development community. This makes the project much more transparent to other developers and potential contributors who will probably have a better grasp of the principles of the BSD, MIT, Apache, MPL, and GPL Licenses, than they would of the Monongahela Copper Mining Institute Database License, v8.3. To the extent that licensing issues are important to contributors, using a license already known to them reduces barriers to entry and will likely make for a more successful project.

For much the same reason, using licenses that are written more clearly and which do not contain ambiguous or unusual terms will also help a project succeed. The BSD and MIT Licenses are models in this regard. The Apache License Version 1.1 is both clear and well-known in the development community, and Version 2.0 is becoming more familiar. The GPL and MPL Licenses, while considerably more complex, are well-written and their principles are well-understood. Developers should avoid licenses that seem ambiguous, unduly confusing, or poorly written.

The most important decision in choosing a license will be the choice between a GPL License and a less restrictive license. A full discussion of the disagreement between the two camps is beyond the scope of this book. However, to put it briefly, the GPL is premised on the belief that non-free software is to be avoided and that free software development projects should be set up to encourage open development models of software development and to discourage reliance on software not developed under an open development model, including all proprietary-licensed software. By requiring that any code developed from or based on GPL-licensed code be GPL-licensed, the GPL creates a strong incentive for programmers to license their code under a GPL License, in the form of access to all the code already GPL-licensed.

The argument in favor of less-restrictive licenses—such as the MIT, BSD, Apache, and MPL Licenses—is that open development model of software development is not inconsistent with the development of software under other models, including proprietary models. The fact that one line of a program, such as the Sendmail program described in Chapter 2, is developed under a proprietary license, does not undermine the open development model, in this view. The more developers and users that are involved in working on particular code, the better, even if some of that development takes place in "closed" development models under proprietary licenses.

In sum, there is no ready answer as to which license is the best for a given project. While a certain license may be better suited for a project, particularly when a substantial amount of work has already been done under that license, such decisions depend largely on circumstances and on the taste of the project developer.

Drafting Open Source Licenses

As should be evident from the previous discussion, drafting a new open source license is probably not the best place to start for most open source projects. In addition to the extra time and expense associated with drafting any legal document, the use of a new license will discourage potential contributors from participating in the project. Those contributors who are concerned about licensing implications will want to read and understand the license. Particularly in the case of long or complex licenses, this may present a substantial barrier to entry.

If you choose to do it, however, the first step in drafting an open source license should be retaining a competent and experienced attorney to undertake the task. While many open source licenses have been drafted by non-lawyers, the drafting of any contract, particularly one with the complexities inherent in open source software licenses, should be undertaken by someone with professional knowledge and experience.

After securing counsel, the next step should probably be devising the basic mechanics of the license. The new author should give serious thought to what the function of the license is intended to be. With open source and free software licenses, the key issue will generally be the generational limitations placed on distribution and modification of the licensed work by licensees. Many of the possible limitations have already been described. The MIT and the BSD Licenses, for example, require only that the text of the license be included in the subsequent distribution and that the required attributions be made. The GPL imposes much more substantial limitations: any distribution or modification of the work by licensees must be consistent with the terms of the GPL. If a licensee wishes to modify and distribute the work, he or she must license future users of that modified work under the GPL. The MPL imposes somewhat similar restrictions for modifications to the licensed work, but it permits either the original or the modified work to be distributed as part of a "Larger Work" under another license, including a proprietary license.

The number of potential variations is nearly infinite. The Open Source Definition, described in Chapter 1, imposes some specific requirements for a license that the author wants to have certified as compliant by the Open Source Initiative.

A brief summary of those requirements follows here. An open source license must permit an open development model to be applied to the licensed work, in that the source code must be provided or otherwise made available with the executable version of the code. The license must permit free modification of the licensed work and free distribution of both the original and the modified work. The license cannot discriminate in its application against any person or group of persons or any field of endeavor.

Of course, a license need not be compliant with the Open Source Definition to be an effective license. But if the intent is to draft an "open source license," failure to comply with the Open Source Definition is a pretty good sign that the drafter is not headed in the right direction. Beyond the fundamentals of the Open Source Definition, there is considerable scope for creativity and ingenuity in drafting licenses.

Many of the licenses described in this book, such as the Apache License, v2.0, and the MPL, begin with long, comprehensive lists of definitions. While not necessary, using such definitions can avoid unnecessary repetition of the same language throughout the license. A definitions section can also avoid accidental, and apparently inconsequential, variations in phrases or sentences that are supposed to be identical. Such variations can lead to potentially serious problems in interpreting the license, as users and contributors, and possibly lawyers, judges, and juries, attempt to determine whether the use of slightly different language was accidental or intentional.

Disclaimer of warranties and limitation of liabilities clauses are virtually universal in open source licenses. While certainly not required by the Open Source Definition, they are prudently included in such licenses to protect the licensor and any potential contributors from liability. Such clauses are not unique to open source licenses—many commercial software licenses contain similar terms.

The use of choice of forum and choice of law clauses is relatively uncommon in open source licenses, but there are many situations in which such clauses could be advantageous to the licensor, particularly for "developer-centric" licenses, such as the Apache License, v2.0, and the Perl License. With such licenses, it is anticipated that the project will remain primarily under the control of its initial developer. That developer may want to choose a local forum and the application of local law for the convenience of the developer in the event any dispute arises under the terms of the license. For example, a developer located in Boston may want to identify the Massachusetts state courts located in Boston as the forum for any dispute under the license and for Massachusetts law to control the interpretation and enforcement of the license. When considering the use of such clauses, developers should consult with a lawyer to make sure that the law that they are choosing to govern the license will interpret and enforce the license consistent with the developer's understanding. Laws vary significantly among different locales: it is certainly possible that a New York court would reach a different conclusion than a Massachusetts court as to how a contract should be interpreted.

One final area that a developer should give some thought to addressing is the applicability of patents to the licensed work. In order to prevent patent litigation to the extent possible, it is probably worthwhile to include a clause in the license that grants specific permission for users to exercise a royalty-free right to any patents held by the licensor, and, depending on the terms of the license, any subsequent contributors.

For a list of licenses that the Open Source Initiative has approved as conforming to their expectations of open source, and for information about their process for approving licenses, visit *http://opensource.org/licenses/*.