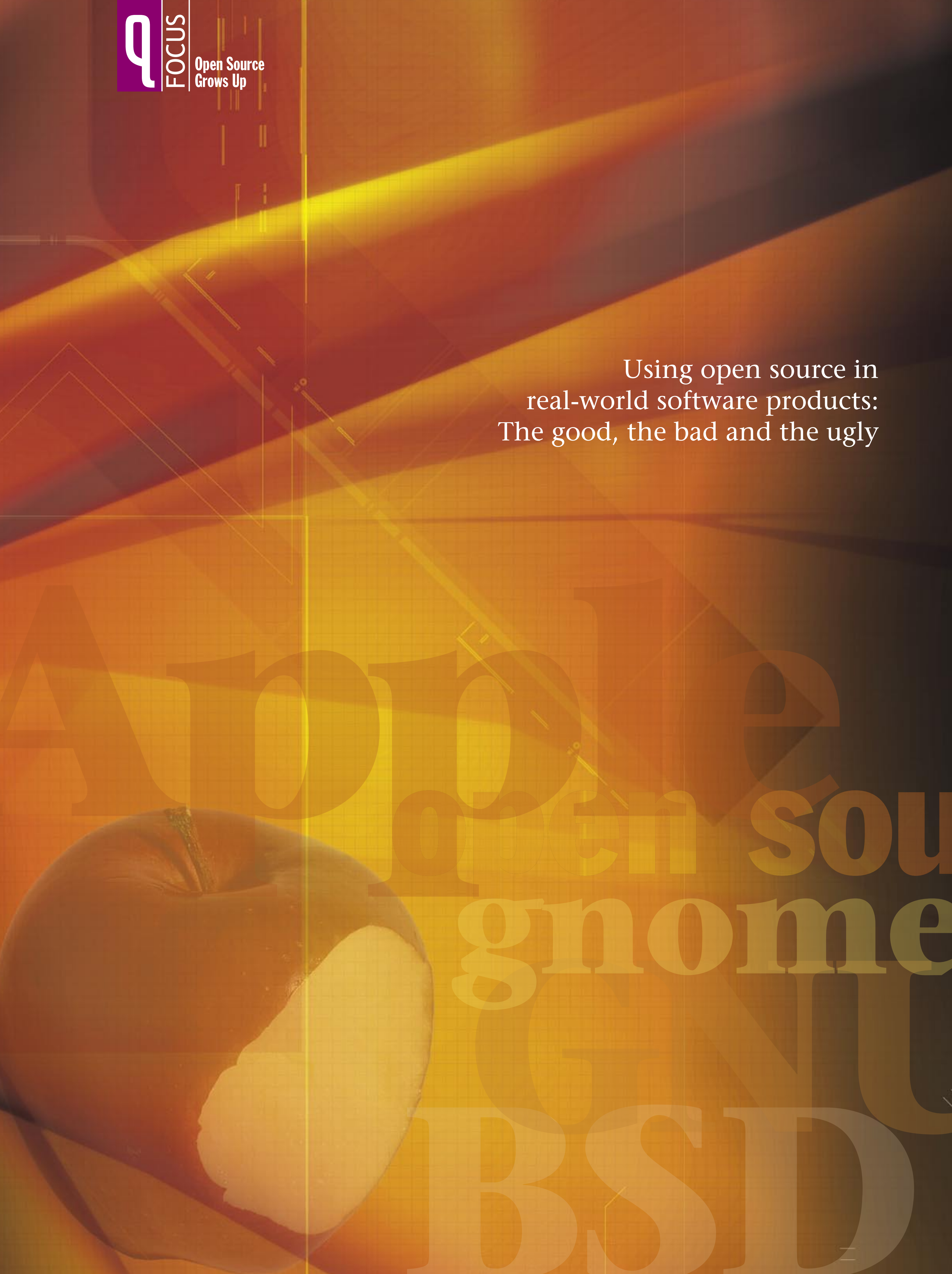
Using open source in real-world software products: The good, the bad and the ugly



Open Source to the Core

JORDAN HUBBARD, APPLE COMPUTER

The open source development model is not exactly new. Individual engineers have been using open source as a collaborative development methodology for decades. Now that it has come to the attention of upper and middle management, however, it's finally being openly acknowledged as a commercial engineering force-multiplier and important option for avoiding significant software development costs.

To put it another way, what object-oriented programming often promises in terms of encouraging "code re-use," open source software is definitely delivering. This does not come without certain costs and potential pitfalls, of course. This article describes the open source adoption process at a typical commercial software operation and discusses some of the more important checklist items that any evaluation of open source as an engineering option should include.

Roughly speaking, this checklist consists of the following: *investigation*, *evaluation*, *adoption*, and *communication*.

INVESTIGATION

There is a lot of open source software in the world, and it is increasing day by day, so the first step in determining whether it can help your project is to make a careful study of what is already available. You don't want to reinvent some expensive wheels.

References to how Apple Computer has used various bits of open source have been included in the following examples to show how one or more pieces of software

Open Source to the Core

have found “institutional relevance.” Examples of corporate usage of open source software abound, however, and an important part of the investigative process for any open source software product involves seeing who else is using it and why. Such examples can serve as a valuable roadmap for your own efforts.

Some well-known sources of open source software:

THE UNIVERSITY OF CALIFORNIA, BERKELEY

In the late 1970s, the Computer Systems Research Group at the University of California, Berkeley, began its seminal research into operating systems and the then-considerable task of extending the rather youthful Unix operating system into something more generally useful. The result of this work was the Berkeley Software Distribution, or BSD, release of Unix, well known for its strong networking capabilities.

Not confined to operating systems, the BSD world also contains a wide selection of libraries, tools, and generally useful code that is distributed under the very liberal BSD license. This license essentially allows code to be used for almost any purpose, closed or open source, and without “encumbering” licensing effects for third-party code that uses it. The original BSD distributions spawned a number of independent projects such as FreeBSD, NetBSD, and OpenBSD, all of which continue to do innovative work that is released under the same BSD license. This has made BSD software a popular choice for commercial software and hardware vendors, who are thus free to create closed source variants of their products without undue restrictions. Such products include Apple’s Mac OS X operating system, whose commands, libraries, and portions of the kernel are heavily based on FreeBSD. A number of commercial network appliances (routers, firewalls, servers) and hardware devices also use the highly cross-platform BSD operating system in an embedded operating system role.

THE APACHE SOFTWARE FOUNDATION

Probably best known for its acclaimed Apache Web server, the Apache Software Foundation (ASF) has since broadened its mandate to cover Java development tools—particularly where Web-based application (servlet) development is concerned—and Java-based build tools. ASF products are widely considered to be best-of-breed and have essentially taken over the Web server market with a market share far exceeding that of anyone else. Apple includes both Apache and Apache2 with Mac OS X.

Software produced by the ASF is released under the Apache license, a spiritual cousin to the BSD license, which allows both closed and open source usage with very few restrictions. See the evaluation section later in this article for more detail.

THE GNU PROJECT

The Free Software Foundation launched the GNU project in 1984 with the initial aim of creating a complete operating system environment (the GNU system). It may not have succeeded in creating a mainstream operating system, but it did create some excellent tools along the way. Among these are the Emacs editor, the GCC (GNU C Compiler), and the GDB (GNU Debugger). The latter two have essentially become de-facto standards in their own right. The GNU project has also created a number of general-purpose libraries and tools for easing the process of creating cross-platform software. The libraries and tools can substantially reduce the amount of work involved in many kinds of software development. Apple ships a fair amount of GNU software with its Mac OS X operating system, including but not limited to GCC, GDB, make, Emacs, and bash.

The greatest caveat to using software from the GNU project is probably its licensing terms. GNU software is released predominately under the GPL (GNU General Public License), with some of its software released under the less-restrictive but still formidable LGPL (GNU Lesser General Public License). Anyone interested in incorporating GPL- or LGPL-licensed software in their own products should certainly read the section on evaluating licenses in this article.

THE GNOME PROJECT

Started in the mid-1990s as a project to bring an advanced desktop environment to the Linux operating system, GNOME (GNU Object Model Environment) has since evolved into a serious source of numerous general-purpose tools and libraries for doing everything from parsing XML to processing audio data. The GNOME project tends to take a very high-level approach to problem solving in general, and this shows in the software it offers. Apple includes GNOME software such as libxml2, an XML parsing library, with Mac OS X.

Most, if not all, of the GNOME project's offerings are licensed under the GPL or LGPL, so the same caveats apply.

THE KDE PROJECT

Another project started in the mid-1990s with the objective of bringing a sophisticated desktop environment to the Linux OS, the KDE Desktop Environment has also created a wide array of general-purpose tools and libraries in the process. These include a Web browser and office productivity suite, as well as libraries for image processing and parsing HTML. Apple includes the latter library, known as KHTML, in its Safari Web browser.

Most, if not all, of the KDE project's offerings are licensed under the GPL or LGPL, so the same caveats apply.

SOURCEFORGE

Billing itself as "the world's largest open source development site," with more than 74,000 hosted projects at the time of this writing, SourceForge is probably not practicing mere hyperbole. If you can imagine it, you can probably find it on SourceForge. Access to each project's information and other resources is made relatively easy. If there are any caveats to using SourceForge, it's simply the sheer magnitude of the material offered to search. It's probably also fair to say that of those 74,000-plus projects, a good number are either moribund or have not made much progress beyond coming up with a nifty project name and creating a SourceForge project. Still, free is free, and SourceForge is definitely worth a stop on any research expedition.

SourceForge projects are released under a variety of licenses, so each should be researched individually as part of your evaluation.

GOOGLE AND SLASHDOT

Though somewhat more scattershot than the other options, simple keyword searches on search engines like Google are often very good ways of pulling out of the air references to what would often otherwise be highly obscure projects. You obviously need to have a fairly clear idea of what you're searching for, but search strings such as "`qsort algorithm`" or "`3D library`" can generate surprisingly relevant results in no time at all.

If you have a little more time on your hands and a willingness to wade through innumerable news articles on everything from the Mars landings to pending court cases on encryption, the Slashdot news site covers ongoing developments in the Apple, BSD, and Linux communities and can be a good source of information you might otherwise miss or not know to look for.

EVALUATION

So, you've done your initial investigation and you've located some open source software that promises to meet your needs. The next thing on your checklist is obviously to evaluate that software in more depth to see if it's genuinely suitable.

LICENSES

The first item on the evaluation checklist, as was already mentioned briefly, is the license. Some open source software licenses come with significant constraints, and for any commercial entity contemplating the use of open source software in a commercial product, review by a competent legal authority is a must. The various nutshell summaries provided here are intended only to give a rough notion of what each license implies, and none is by any means a substitute for a legal review of the actual license text, should you decide to proceed with the software in question.

The BSD License. This is one of the more liberal licenses. The original version from the University of California consists of three clauses, roughly summarized as:

1. Don't remove our copyright notice and disclaimer from the source code.
2. If you ship binaries, include the copyright notice and disclaimer somewhere with the binaries.
3. You can't use our name to endorse or advertise your product unless you get permission in writing first.

Some BSD projects, such as FreeBSD, have even removed the third clause from their own version of this license, essentially asking only for some credit for their work and disclaiming any responsibility for its use or misuse. As licenses go, lawyers tend to rank this one near the top of their personal preference lists.

MIT Consortium License. Also sometimes known as the X11 license, the MIT Consortium license is extremely liberal. It is also a very short license. In essence, it gives you full permission to use the software for any purpose, just as long as you don't sue the authors if it breaks or use their names in your advertising. It is generally considered "morally equivalent" to the BSD license, though it places even fewer restrictions on software licensed under it.

The Apache Software Foundation License. The ASF license is very similar to the original BSD license but with two additional clauses, both essentially intended to strengthen the point that you're not allowed to use the terms *Apache* or *Apache Software Foundation* to endorse or promote your own products, or use those terms in any derived works, without prior written permission from the ASF. The basic intention here appears to prevent dilution of the Apache brand or have it used in situations that the ASF board would not approve. If you use Apache code in your product but call your product something else, you're fine.

The ASF is revising this license and may be done by the time this article is published. Those who are interested should definitely visit the Apache Web site for more information on updates to this (or any other) license.

The GNU GPL/LGPL Licenses. The GNU licenses definitely require serious legal review, given that they come with significant strings attached. Roughly summarized, those strings are:

GPL. You are allowed to use GPL'd software in your own code as long as your own code is also licensed under the GPL and provided under the same terms (basically free of charge and in source form) to end users.

LGPL. You are allowed to link with libraries that are LGPL-licensed without having your code under the same license (hence, the *Lesser*, since it is less restrictive). This does not mean that you can freely "borrow" code out of LGPL'd libraries and sprinkle it throughout your own code or that you are free from the requirement of providing sources for the LGPL'd code to end users.

Again, these are extremely condensed versions of these licenses, given that each runs to several thousand words, so be sure to read the full text of each carefully.

Dual-Use Licenses. One of the biggest potential pitfalls in using open source software comes with the dreaded dual-use license. This license allows the software to be used by other open source software but presents something of a "poison pill" to closed source or commercial software. In such cases, a special clause is invoked, which generally requires that the user pay for either a software license or usage rights to some patent under which the software is covered.

Sometimes, even commercial users can get away without having to pay fees for such software if they structure their usage of it so that they can give away their own source that uses it, thus qualifying for the open source definition of the license. This is tricky, however, particularly when libraries are involved. If you're an operating system vendor and you ship a library for which a dual-license exists, making sure that your clients of this library are provided in source form, what about your customers who may link their own products with this library? Clearly, you don't want to be handing your customers a potential landmine they may not even be aware of. This is especially true when discussing software that falls under a patent, which is generally easier to infringe inadvertently.

It bears repeating that there is simply no substitute for a proper legal review of the licenses for any and all open source software you intend to use and, in many cases, the software that it in turn uses. It is quite easy to fall into the trap of thinking that you qualify for a license to use some specific piece of software, only to find upon closer inspection that it in turn requires yet another piece of software for which you do not qualify for a license (at least not without a substantial dollar investment).

Dual-use licenses are a somewhat controversial part of the open source development world, so no specific examples of such software will be given here, but rest assured that they exist.

Also note that software released under dual-use licenses is very different from software that is dual-licensed, such as Perl, Mozilla, and Qt. With dual-licensed software, the user can choose the least restrictive or otherwise most appropriate license. This has become a popular option for some corporations that wish to offer a "commercial version" of their software, generally with full technical support, while still making it freely available to those who are willing to accept the software as is, without such support.

CODE QUALITY

Once you've been through the license, the next important item on the evaluation checklist is that elusive determination of code quality. Sadly, not all open source software is created equal when it comes to quality, and evaluating the general track record and length of time in the field for the project that created it is one useful (though not foolproof) way of gauging how refined its code is likely to be. You should also make some effort to see how actively the code is being maintained (When were the last modifications made? Why?) and how many maintainers the code has had over its lifetime; software that is passed frequently from hand to hand often suffers from the inexperience and learning curve of each new maintainer.

COMMUNITY

Next, you need to assess just how large the user community surrounding the software is. Is there a user community? How responsive do its members appear to be to questions and/or bug reports from others on their mailing lists or Web forums? Are new contributions incorporated in a timely fashion or do they appear to fall on the floor? Code that is actively maintained is obviously far more likely to be secure and functional, so all of these are key factors to be weighed before committing to a piece of open source software. The last thing you need is to inherit a pile of abandon-ware, which you subsequently, and probably painfully, find out was abandoned for good reason.

SECURITY

Another important though sometimes overrated statistic is the number of security advisories that have been posted against the software. What makes this metric tricky is the fact that software that is buggy but not very popular is unlikely to have many security advisories posted against it, simply because there just aren't that many people using (or tempted to attack) it. Conversely, highly popular software gets attacked on a frequent basis and may show up disproportionately in the security advisories. This is a judgment call, obviously, but if a piece of software appears to show up in security advisories with alarming frequency, you may do well to seek equally functional alternatives.

OK, so you've done your investigation, you've found some suitable software, you've evaluated it and found it to be good, so the hard work is behind you, right? Not quite, unfortunately, because the next step is...

ADOPTION

Incorporating open source software code into your product has its own internal costs; it's never truly "free" in that respect. There are some important internal checklist items you should consider in the areas of management, technology, and community. These will help ensure that open source software can and will succeed in your company.

MANAGEMENT: INTERNAL EVANGELISM

Engineers can be surprisingly averse to adopting comparatively finished products rather than developing their own from scratch, even when the latter option constitutes significantly more work. Even if you get past the less justifiable issues of ego and the loss of satisfaction, both of which can be significant barriers in their own right, the engineers are likely to have concerns that are not so easy to dismiss: code quality, the ease with which it can be modified, and expectations the external open source software developers are likely to have of them. Unless your engineers are already highly bullish on the idea of using open source, getting through this part of the process can require a fair amount of dialogue and finesse.

Your marketing people will typically like the buzz-worthiness of using open source but, if open source is to be leveraged heavily, you will probably also need clear points of differentiation that justify the commercial value of your product. Management and the legal department will also need to clearly understand where the code is coming from, what strings are attached to it, and what engineering's ongoing obligations are expected to be.

TECHNOLOGY: PORTING

Open source software code may or may not come ready to run on your platform. Assume that a certain amount of porting and/or adaptation to your needs will be necessary.

COMMUNITY: MANAGEMENT OF PROCESS AND RELATIONSHIPS

As you progress with the code base, donating your changes back to the open source software community is an excellent way to build trust, make future merges with the external open source software code base easier (and less expensive), and help advertise your presence to many opinion leaders in the same community likely to influence purchasing decisions of your product. It is also an important part of establishing a certain amount of "credit" in the barter economy that open source software development often represents. The volunteer engineers in the open source software community are far more likely to help those who have demonstrated their commitment to the success of the overall open source software development process. Even simply keeping lines of communication open, with or without code contributions, is of significant value; most open source software engineers have an active interest in what their code is being used to do.

The importance of doing all of this cannot be overstated. If you establish a reputation, either fairly or unfairly, as a "taker" who has no interest in giving something back or even communicating with the open source software community in good faith, then you'll find yourself at the bottom of a hole you may never manage to climb out of, your reputation sullied and the repercussions of that likely to be far more extensive than you might think. It's a small world where software

Open Source to the Core

engineering is concerned, particularly with everyone now networked, and engineers are clearly part of any high-tech company's lifeblood, so the conclusions to be drawn from this are pretty obvious.

Apple itself has learned some important lessons here, creating additional channels of communication/collaboration, such as the OpenDarwin project, and allowing engineers to communicate more openly whenever it becomes apparent that the more traditional mechanisms for doing so are overly restrictive or otherwise ineffective. A commitment to constantly evaluate and evolve the communications process when necessary also goes a long way.

COMMUNICATION

You've managed to incorporate open source software into your product, you've given the relevant changes back to the open source software community, and established a reasonable rapport with them. Now it's time to deploy your product. What's next on the checklist? Perhaps obvious to some, but, sadly, not to all, is communication, both in the initial stages and later in sustaining your product after deployment.

INITIAL STAGES

Marketing. First and foremost, your marketing people will (or should) want to have a prepared message about your use of open source, even if it's only to respond to any questions that may come up. Make sure that they also know enough to make correct assertions about it, or you may find yourself paying the price on Slashdot when one of them makes an embarrassing public gaffe about who provided the technology or attributes it to someone else.

Engineering. Engineering will also need to understand that there will be people external to the company with their own perceptions, both negative and positive, of your use of open source. How individual engineers interact (or don't) with this external community will have a lot to do with this balance of positive/negative perception going forward, so make sure everyone's on the same page before you go public.

RESOURCES

SOURCES OF OPEN SOURCE SOFTWARE

FreeBSD
http://www.freebsd.org

NetBSD
http://www.netbsd.org

OpenBSD
http://www.openbsd.org

Apache Foundation Software
http://www.apache.org

The GNU Project
http://www.gnu.org

The GNOME Project
http://www.gnome.org

The KDE Project
http://www.kde.org

SourceForge.net
http://sourceforge.net

Google
http://www.google.com

Slashdot
http://slashdot.org

Apple
Open Darwin
http://www.opendarwin.org

Mozillazine
(for information on
the Safari Web browser)
http://Weblogs.mozillazine.org/hyatt/

OPEN SOURCE SOFTWARE LICENSES

BSD
http://www.opensource.org/licenses/bsd-license.php

MIT Consortium
http://www.opensource.org/licenses/mit-license.php

Apache Software Foundation
http://www.apache.org/LICENSE.txt

GNU GPL/LGPL
http://www.gnu.org/licenses/licenses.html#GPL

Legal. Your legal department must understand and drive the basic obligations, if any, that go along with deployment of this software (such as distributing or making available source for GPL code). Don't assume that the engineering people will have the bandwidth or the understanding to do this on their own—someone from legal should make sure this really happens.

SUSTAINING AFTER DEPLOYMENT

Truly leveraging open source software also means understanding and acknowledging that engineers external to the company can be as important to your current and future deliverables as engineers inside the company. That's a concept that many companies find hard to grasp, given the rather stark difference in how much they control the work habits of each. It is, nonetheless, true on the balance, and a number of factors need to be considered in managing your development process going forward, after your product has shipped:

NDA (nondisclosure agreement) issues and corporate security can certainly make communication with the open source software community tricky, but it's still manageable with proper guidelines in place. A far greater and more pervasive problem is one where engineers simply choose not to communicate externally at all because of the amount of work involved and/or deadline pressures that drive the focus of their communication inward. Making it clear just where and when they are allowed to communicate and "expected" to communicate can go a long way toward alleviating this problem and prevent alienation from the open source software community.

Soliciting and understanding the open source software community's goals for the product(s) you've incorporated is one important part of protecting your own future interests. Defining and trying to communicate your own needs for the product to the open source software community will also certainly influence their own direction, and openly declaring your own intentions also helps build bridges to the community since it's clear that you're making some effort to keep them in the loop as peers, not simply as sources of free labor.

Managing transitions and sustaining engineering is also a key item for post-deployment, since healthy open source software does not stand still by any means, whereas commercial product cycles can be rather lengthy. You must project and factor in synchronization costs into your timeline.

Finally, the open source software community can be an invaluable resource when it comes to recruiting skilled, motivated engineers who come with a ready-built understanding of at least some aspect of your product. This is one reason why developing and sustaining relations with the open source software community at the outset is so critical, and it should also be understood that open source software engineers recruited from this community will have a residual loyalty to this community. One nice way of maintaining your relationship with the community is to allow these engineers to continue doing some amount of (now-subsidized) work on the open source software public code base. They get better code, you get better code, and both sides win.

A SOCIAL ORDER AND ENGINEERING DISCIPLINE

Open source is not just a way of getting free code; it is both a social order and an entire engineering discipline in its own right, with informal "rules of the road" that must be respected if you are to be at all successful there. It is, however, also one of the most rewarding ways of doing software engineering today, allowing small engineering teams to tackle the sorts of challenges that would have been considered simply impossible for groups of their size before. It has also greatly expanded what were formerly engineering "island communities" at both small and large companies—communities that were highly insular and lacked sufficient exposure to many potentially better ways of doing such work more cheaply and with higher quality. One company could not possibly hire all the best engineers in the world, but by actively involving itself in the open source community, it no longer has to! Q

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

JORDAN HUBBARD is the manager of BSD technologies for Apple's core OS engineering team. He oversees the BSD technology base for Darwin, the Unix-based core of Mac OS X. Before joining Apple in 2001, Hubbard was a principal technologist for Wind River Systems, where he was responsible for the FreeBSD CD-ROM product line. He is a cofounder of the FreeBSD project, which began in 1992. Hubbard began his career in software in the 1970s, working on minicomputers, and has held various engineering and management positions in organizations, including U.C. Berkeley and Digital Equipment Corporation. He is a frequent contributor to the open source community and has been writing free software since 1982, beginning with Volume 1 of the comp.sources.unix archive and continuing with various works on MIT's X Contributed Software collection.