9 Making Lightning Strike Twice

Charles B. Weinstock and Scott A. Hissam

The Software Engineering Institute (SEI) is a federally funded research and development center (FFRDC) that is operated by Carnegie Mellon University and sponsored by the U.S. Department of Defense (DoD). One of our many activities is to advise the DoD on software-related issues. Several years ago, a new silver bullet arrived with the words "open-source software" (OSS) emblazoned on its side. As OSS became more prevalent, we were asked to determine its applicability to DoD systems—was it really a silver bullet? To answer these questions, we undertook a study of what OSS is, how it is developed, and how it is contributing to the way we develop software. In particular, we wanted to learn where and how OSS fit into the general practice of software engineering. The study attempted to identify OSS from a practical perspective, with the goal of differentiating between hype and reality. To this end, we conducted interviews, participated in open-source development activities, workshops, and conferences, and studied available literature on the subject. Through these activities, we have been able to support and sometimes refute common perceptions about OSS.

Perceptions of OSS

It is not surprising, given the attention that OSS has received, that there are myths about OSS—both positive and negative. In this section, we'll look at some of the myths.

Myth: OSS, being under constant peer review by developers around the world and around the clock, must therefore be of higher quality; that is, it must be more reliable, robust, and secure than other software.

Raymond (2001) argues that OSS developers write the best code they can possibly write because others will see the code. He also asserts Linus's Law:

"Given enough eyeballs, all bugs are shallow (p. 41)." Because there are thousands of developers reviewing OSS code, a flaw in the code will be obvious to someone.

In fact there *is* open-source software that is good software and, by many measures, high-quality software (Linux and Apache, to name but two). Does all OSS shares this status?

Myth: Having the source code for OSS gives more control because of the ability to read and modify the source code at will.

This myth is viewed as the main advantage that OSS has over closed-source software (CSS), where one is at the mercy of the vendor. If the vendor should go out of business (or otherwise stop supporting the software), the user has no recourse. With OSS, there is no vendor to go out of business. We'll explore the relationship of OSS to CSS further in a later section.

Myth: OSS has poor documentation and little support.

The assumption is that hackers are off coding wildly and have neither the time nor the motivation to document what they produce. The concern that there is little support comes from the sense that there is no one to phone when there is a problem. O'Reilly (1999) discusses this myth briefly. There is a trend towards gaps in support and/or documentation being filled by support companies (e.g., Red Hat). Does this apply to all OSS?

Myth: There are armies of programmers sitting around waiting and eager to work on an OSS project free of charge, making it possible to forego the traditional development costs associated with traditional software-development activities.

The old adage "You can lead a horse to water, but you can't make him drink" best describes the OSS community—that is, "You can put the code out in the community, but you can't make a hacker code." The likelihood that an OSS product will be successful (or that the hackers will help you) is based on the characteristics of that product.

Myth: OSS hackers are a group of mavericks working in an unorganized, haphazard, ad hoc fashion.

Given the global reach of the Internet and the therefore distributed nature of hacker-based development, this might seem to be an obvious conclusion. For some, this is the allure of the OSS development process—that there is no "process-monger" or program manager hanging over the progress of the development effort (hence the process is unpredictable and progress is immeasurable). We refute this myth in the following Apache case study.

Case Studies

One of the ways in which we attempted to understand the OSS phenomenon was to get involved in or research several efforts/events. There were five such studies, each giving us a different perspective of OSS, in terms of software development, the products themselves, and users:

- AllCommerce—an e-commerce storefront solution
- Apache—an open-source Web server
- Enhydra—a Java-based application server
- NAIS—a NASA-operated Web site that switched from Oracle to MySQL
- Teardrop—a successful Internet attack affecting OSS and CSS

The purpose in selecting these specific OSS projects was to take varying perspectives of OSS, in terms of software development, the products themselves, and users.

The AllCommerce case study focused on software development in the OSS paradigm. A member of the SEI technical staff got involved in the process of hacking the product to discover bugs and add new features to the product. The express purpose of this case study was to obtain firsthand experience in working on an OSS product from the inside; that is, to learn the process by which changes are actually proposed, tracked, selected/voted on, and accepted. We learned that while it is fairly easy to have an impact on OSS, the OSS project needs a critical mass to stay alive. This can happen because there are many people interested in the project (for whatever reason) or because of the existence of a serious sponsor. In the case of AllCommerce, development progressed only when it had a sponsor that provided support in terms of employee time and other resources. When that sponsor folded, AllCommerce for all intents and purposes went dormant, and as of this writing remains so.

The Apache case study took an academic, research perspective (actually the result of a doctoral thesis) of the OSS-development process. This case study looked at the individual contributions made to the Apache Web server over the past five years and examined whether that contributor was from core or noncore Apache developers. From this study we learned that the core developers hold on to control of what goes into Apache and what does not. As a result the development process for Apache ends up being very similar to the development process of a good commercial software vendor (Hissam et al. 2001).

From a purely product-centric perspective, the Enhydra case study focused on the qualitative aspects of an OSS product and looked at coding

problems found in the product by conducting a critical code review. We learned that claims to the contrary notwithstanding, the Enhydra source code is no better than commercial source code we have reviewed in the past. The code as a whole is not outstanding, but it is not terrible, either; it is simply average. It appears in this case that the many eyes code-review assertion has not been completely effective, given that our review was casual and tended to look for common coding errors and poor programming practices.

The NAIS case study, which focused on the end user, looked at a real application developer who switched from a commercially acquired software product to an OSS product. Specifically, this case study examined how and why that particular OSS product was selected, the degree to which the application developer was engaged with the OSS development community, and the level of satisfaction that the NAIS had with the selected OSS product. They chose MySQL to replace an Oracle database that they could no longer afford and have been quite happy with the results.

Finally, the Teardrop case study looked into one of the predominant assertions about OSS: that OSS is more secure than software developed under more traditional means. This case study takes apart one of the most successful distributed denial-of-service (DDoS) attacks and looks at the role that OSS played in the propagation of that attack on CSS and the response by the OSS community. The code that was exploited to conduct this attack had a problem that was easily fixed in the source code. At the same time another problem, which had not yet been exploited was also fixed. This was fine for the Unix systems this code ran on. It turns out, though, that Microsoft Windows shared the same flaws and only the first of them was fixed on the initial go around. Attackers noted the fix in the Unix code that tipped them off to a problem that they were ultimately able to use against Windows—until it too was fixed (Hissam, Plakosh, and Weinstock 2002).

From this study we learn that OSS is not only a viable source of components from which to build systems, but also that the source code enables the integrator to discover other properties of the component that are not typically available when using CSS components. Unfortunately there is a cost to this benefit, as cyber terrorists also gain additional information about those components and discover vulnerabilities at a rate comparable to those looking to squash bugs.

This is not to say that security through obscurity is the answer. There is no doubt that sunshine kills bacteria. That is, the openness of OSS development can lead to better designs, better implementations, and eventually

better software. However, until a steady state in any software release can be achieved, the influx of changes, rapid release of software (perhaps before its time), and introduction of new features and invariably flaws will continue to feed the vicious cyclic nature of attack and countermeasure.

What It Takes for a Successful OSS Project

The success of an open-source project is determined by several things that can be placed loosely into two groups: people and software. For instance, the development of an OSS accounting system is less likely to be successful than that of a graphics system. The potential developer pool for the former is much smaller than that for the latter—just because of interest. Paraphrasing Raymond (2001), "The best OSS projects are those that scratch the itch of those who know how to code." This says that a large potential user community is not by itself enough to make an open-source project successful. It also requires a large, or at least dedicated, developer community. Such communities are difficult to come by, and the successful project is likely to be one that meets some need of the developer community.

The success stories in OSS all seem to scratch an itch. Linux, for instance, attracts legions of developers who have a direct interest in improving an operating system for their own use. However, it scratches another important itch for some of these folks: it is creating a viable alternative to Microsoft's products. Throughout our discussions with groups and individuals, this anti-Microsoft Corporation sentiment was a recurring theme.

Another successful open-source project is the Apache Web server. Although a core group is responsible for most of its development, it is the Web master community that actually contributes to its development.

On the other hand, as we saw in the AllCommerce case study, without serious corporate sponsorship AllCommerce was unable to sustain itself as a viable open-source project. Without being paid, there weren't enough developers who cared deeply enough to sustain it.

Although people issues play a large part in the success of an open-source project, there are software issues as well. These issues can be divided into two groups as well: design and tools.

The poorly thought out initial design of an open-source project is a difficult impediment to overcome. For instance, huge, monolithic software does not lend itself very well to the open-source model. Such software

requires too much upfront intellectual investment to learn the software's architecture, which can be daunting to many potential contributors. A well-modularized system, on the other hand, allows contributors to carve off chunks on which they can work.

At the time we conducted our study, an example of an open-source project that appeared to work poorly because of the structure of the software was Mozilla (the open-source Web browser). In order to release Mozilla, Netscape apparently ripped apart Netscape Communicator, and the result, according to some, was a "tangled mess." Perhaps it is not coincidental that until recently, Mozilla had trouble releasing a product that people actually used.

To its credit, Netscape realized that there was a problem with Mozilla and, in an attempt to help the situation, created a world-class set of open-source tools. These tools, such as Bonsai, Bugzilla, and Tinderbox, support distributed development and management and helped developers gain insight into Mozilla. While perhaps not true several years ago, the adoption of a reasonable tool base is required for an open-source project to have a significant chance of success (if only to aid in the distributed-development paradigm and information dissemination). Tools such as revision-control software and bug-reporting databases are keys to success. Fortunately for the community, organizations like SourceForge (http://www.sourceforge.net) are making such tool sets easily available; this goes a long way towards solving that aspect of the problem.

A final factor in the success of an open-source project is time. Corporate software development can be hampered by unrealistically short time horizons. OSS development can be as well. However in the former case, projects are all too often cancelled before they have a chance to mature, while in the latter case an effort can continue (perhaps with reduced numbers of people involved). The result may be that an apparent failed open-source project becomes a success. Because of this it is difficult to say that a particular project has failed. Examples of OSS projects that appeared to have failed yet now seem to be succeeding include GIMP (Photoshop-like software) and the aforementioned Mozilla. "It hasn't failed; it just hasn't succeeded—yet."

The OSS Development Model

It might not be surprising that the development process for OSS differs from traditional software development. What might be surprising to some is how ultimately similar they are.

Traditional software development starts with a detailed requirements document that is used by the system architect to specify the system. Next comes detailed system design, implementation, validation, verification, and ultimately, maintenance/upgrade. Iteration is possible at any of these steps. Successful OSS projects, while not conducted as traditional (e.g., commercial) developments, go through all of these steps as well.

But the OSS development model differs from its traditional perhaps not-so-distant cousin. For instance, requirements analysis may be very ad hoc. Successful projects seem to start with a vision and often an artifact (e.g., prototype) that embodies that vision—at least in spirit. This seems to be the preferred way of communicating top-level requirements to the community for an OSS project. As the community grows, the list of possible requirements will grow as well. Additional requirements or new features for an OSS project can come from anyone with a good (or bad) idea. Furthermore, these new requirements actually may be presented to the community as a full-fledged implementation. That is, someone has what he thinks is a good idea, goes off and implements it, and then presents it to the community. Usually this is not the case in a traditional project.

In a traditional project, the system architect will weigh conflicting requirements and decide which ones to incorporate and which to ignore or postpone. This is not done as easily in an OSS development effort, where the developer community can vote with its feet. However successful projects seem to rely on a core group of respected developers to make these choices. The Apache Web server is one example of such a project. This core group is taking on the role of a system architect. If the core group is strong and respected by the community, the group can have the same effect (virtually identical) as determining requirements for a traditional development effort.

Implementation and testing happens in OSS development efforts much as it does for traditional software–development efforts. The main difference is that these activities are often going on in parallel with the actual system specification. Individual developers (core or otherwise) carve out little niches for themselves and are free to design, implement, and test as they see fit. Often there will be competing designs and implementations, at most one of which will be selected for inclusion in the OSS system. It is the core group (for systems so organized) that makes the selections and keeps this whole process from getting out of control.

Finally, to conduct maintenance activities, upgrade, re-release, or port to new platforms, the open-source community relies on sophisticated tools for activities such as version control, bug tracking, documentation

maintenance, and distributed development. The OSS project that does not have or use a robust tool set (usually open source itself) either has too small a community to bother with such baggage or is doomed to failure. This is also the case for traditional development.

The Relationship of OSS to CSS

Judging from the press it receives, OSS is something new in the world of software development. To the limited extent that the press itself is sensitive to the term, there is truth to that statement. It would be fair to acknowledge that more people (and not just software engineers) are now sensitive to the term *open source* than ever before—for which we can also thank the press. But what makes OSS new to the general, software systems engineering community is that we are faced with more choices for viable software components than ever before. But you may ask yourself, before what?

The World Before OSS

Before OSS became a popular term, software engineers had three generalized choices for software components:

- The component could be built from the ground up.
- The component could be acquired from another software project or initiative.
- The component could be purchased from the commercial marketplace.

If the component were to be built from the ground up, there were basically two approaches: to actually undertake the development of the component from within the development organization (i.e., inhouse), or to negotiate a contract to develop the component via an external software-development organization. Essentially the component was custom-built. As such, the software sources were available for the component acquired in this fashion.

Another approach was to locate components of similar functionality from other (potentially similar) software projects. The term often used in this context was *reuse* or *domain-specific reuse*. If a component could be located, it could then be adapted for the specific needs of the using software-development activity. In U.S. government vernacular, this was also referred to as government off-the-shelf (GOTS) software. Typically, reuse libraries and GOTS software came in binary and source-code form.

Finally, software engineers had the option of looking to the commercial marketplace for software components. Software-development organizations would undergo market surveys trying to locate the components that best fit their needs. Evaluations would commence to determine which of the commercial offerings most closely matched and a selection would be made. In many instances, the source code was not delivered as part of the component's packaging. In some cases, the source code may have been available for an additional cost (if at all). And in the event that the source code could be bought, there were (and still are) very restrictive limitations placed on what could and could not be done to those sources.

The World after OSS

With the advent of OSS, the community has an additional source of components, which is actually a combination of all three of the choices listed earlier. OSS and reusable components are very much alike in that they are both developed by others and often come in binary and source-code form. But like reusable components, it can be challenging to understand what the OSS component does (Shaw 1996).

Because it comes with the source, OSS is similar to custom-built software. However, it lacks the design, architectural, and behavioral knowledge inherent to custom-built software. This is also a problem with commercially purchased software. This lack of knowledge allows us to draw a strong analogy between OSS and COTS software in spite of the source code being available for the former and not for the latter.

The SEI has been studying COTS-based systems for a number of years and has learned some important lessons about them, many of which apply directly to OSS.[1]

Organizations adopting an OSS component have access to the source, but are not required to do anything with it. If they choose not to look at the source, they are treating it as a black box. Otherwise they are treating it as a white box. We discuss both of these perspectives in the following sections.

OSS as a Black Box

Treating OSS as a black box is essentially treating it as a COTS component; the same benefits and problems will apply. For instance, an organization adopting COTS products should know something about the vendor (e.g., its stability and responsiveness to problems), and an organization adopting OSS should know something about its community.

If the community is large and active, the organization can expect that the software will be updated frequently, that there will be reasonable quality assurance, that problems are likely to be fixed, and that there will be people to turn to for help. If the community is small and stagnant, it is less likely that the software will evolve, that it will be well tested, or that there will be available support.

Organizations that adopt COTS solutions are often too small to have much influence over the direction in which the vendor evolves the product (Hissam, Carney, and Plakosh 1998). Black-box OSS is probably worse in this regard. A COTS component will change due to market pressure, time-to-market considerations, the need for upgrade revenue, and so forth. OSS components can change for similar market reasons, but can also change for political or social reasons (factions within the community), or because someone has a good idea—though not necessarily one that heads in a direction suitable to the organization.

Organizations that adopt COTS products can suffer from the vendor-driven upgrade problem: the vendor dictates the rate of change in the component, and the organization must either upgrade or find that the version it is using is no longer supported. This same problem exists with OSS. The software will change, and eventually the organization will be forced to upgrade or be unable to benefit from bug fixes and enhancements. The rate of change for an eagerly supported OSS component can be staggering.

Organizations that adopt COTS solutions often find that they have to either adapt to the business model assumed by the component or pay to have the component changed to fit their business model (Oberndorf and Foreman 1999). We have found that adapting the business model usually works out better than changing the component, as once you change a component you *own* the solution. If the vendor does not accept your changes, you'll be faced with making them to all future versions of the software yourself.

For black-box OSS, it may be easier for a change to make its way back into the standard distribution. However, the decision is still out of the organization's control. If the community does not accept the change, the only recourse is to reincorporate the change into all future versions of the component.

Because of a lack of design and architectural specifications, undocumented functionality, unknown pre- or post-conditions, deviations from supported protocols, and environmental differences, it is difficult to know how a COTS component is constructed without access to the source code. As a consequence, it can be difficult to integrate the component. With OSS,

the source is available, but consulting it means that the component is no longer being treated as a black box.

OSS as a White Box

Because the source is available, it is possible to treat OSS as a white box. It therefore becomes possible to discover platform-specific differences, uncover pre- and post-conditions, and expose hidden features and undocumented functionality. With this visibility comes the ability to change the components as necessary to integrate them into the system.

However sometimes the source is the only documentation that is provided. Some consider this to be enough. Linus Torvalds, the creator of Linux, has been quoted as saying, "Show me the source" (Cox 1998). Yet if this were the case, there would be no need for Unified Modeling Language (UML), use cases, sequence diagrams, and other sorts of design documentation. Gaining competency in the OSS component without these additional aids can be difficult.

An organization that treats OSS as a white box has a few key advantages over one that treats it as a black box. One advantage is the ability to test the system knowing exactly what goes on inside the software. Another advantage is the ability to fix bugs without waiting for the community to catch up. A seeming advantage is the ability to adapt the system to the organization's needs. But as already discussed, the rejection of your change by the community means that you own the change and have given up many of the benefits of OSS.

Acquisition Issues

According to the President's Information Technology Advisory Committee (PITAC): "Existing federal procurement rules do not explicitly authorize competition between open-source alternatives and proprietary software. This ambiguity often leads to a de facto prohibition of open-source alternatives within agencies" (PITAC 00, 6).

The PITAC recommends that the federal government allow open-source development efforts to "compete on a level playing field with proprietary solutions in government procurement of high-end computing software." We wholeheartedly endorse that recommendation.

In the presence of such a level playing field, acquiring OSS would not be fundamentally very different from acquiring COTS software. The benefits and risks would be similar and both must be judged on their merits.

We've already discussed issues such as security in the open-source context, so we won't consider them here. Those sorts of issues aside, there are two risks that an organization acquiring OSS faces:

- That the software won't exactly fit the needs of the organization
- That ultimately there will be no real support for the software

We'll address each of these in turn.

A key benefit of OSS is that the sources are available, allowing them to be modified as necessary to meet the needs of the acquiring organization. While this is indeed a benefit, it also introduces several significant risks. Once the OSS is modified, many open-source licenses require the organization to give the changes back to the community. For some systems this might not be a problem, but for others, there might be proprietary or sensitive information involved. Thus it is very important to understand the open-source license being used.

As discussed in the preceding section on CSS, just because a modification is given back to the community does not mean that the community will embrace it. If the community doesn't embrace it, the organization faces a serious choice. It can either stay with the current version of the software (incorporating the modifications) or move on to updated versions—in which case, the modifications have to be made all over again. Staying with the current version is the easy thing to do, but in doing so, you give up some of the advantages of OSS.

With COTS software there is always the risk of the vendor going out of business, leaving the organization with software but no support. This can be mitigated somewhat by contract clauses that require the escrowing of the source code as a contingency. No such escrow is needed for OSS. However, in both cases, unless the organization has personnel capable of understanding and working with the software's source code, the advantage of having it available is not clear. Certainly there would be tremendous overhead should there be a need to actually use the source code; by taking it over, you are now essentially in the business of producing that product.

Most government software is acquired through contracts with contractors. A contractor proposing an open-source solution in a government contract needs to present risk-mitigation plans for supporting the software, just as it would have to do if it were proposing a COTS product. In the case of a COTS product, this might include statements regarding the stability of the vendors involved. No such statement is valid regarding OSS. The community surrounding an open-source product is not guaranteed to

be there when needed, nor is it guaranteed to care about the support needs of the government. Furthermore, if the proposing contractor is relying on the OSS community to either add or enhance a product feature or accept a contactor-provided enhancement in the performance of the government-funded software-development contract, the government should expect a mitigation if the OSS community does not provide such an enhancement or rejects the enhancement outright. Thus the ultimate support of the software will fall on the proposing contractor.

Security Issues

There are, of course, unique benefits of OSS—many of which have been discussed elsewhere in this book. From an acquisition point of view, the initial cost of OSS is low. Also, at least for significant open-source products, it is likely (but by no means guaranteed) that the quality of the software will be on a par with many COTS solutions. Finally, when modifications are needed, it is guaranteed that they can be made in OSS. For COTS software, there is always the possibility that the vendor will refuse. (But, as we've seen, the ability to modify is also a source of risk.)

Trust in the software components that are in use in our systems is vital, regardless of whether the software comes from the bazaar or the cathedral. As integrators, we need to know that software emanating from either realm has been reviewed and tested and does what it claims to do. This means that we need eyes that look beyond the source code and look to the bigger picture. That is the holistic and system view of the software—the architecture and the design.

Others are beginning to look at the overall OSS development process (Feller and Fitzgerald 2000; Nakakoji and Yamamoto 2001; Hissam et al. 2001). More specifically, from the Apache case study (discussed earlier), we observed what type of contributions have been made to the Apache system and whether those who made them were from core or noncore Apache developers. We learned that a majority (90 percent) of changes to the system (implementation, patches, feature enhancements, and documentation) were carried out by the core-group developers, while many of the difficult and critical architectural and design modifications came from even fewer core developers. Noncore developers contributed to a small fraction of the changes. What is interesting is that the Apache core developers are a relatively small group compared to the noncore developers—in fact, the size of the core developers is on a par with the typical size of development teams found in CSS products.

This is not intended to imply that OSS lacks architectural and design expertise. Actually, the Apache modular architecture is likely central to its success. However, even with the existence of a large community of developers participating actively in an OSS project, the extent that many eyes are really critiquing the holistic view of the system's architecture and design, looking for vulnerabilities is questionable.

This is an issue not just for OSS: it is a problem for CSS as well. That is, in CSS we have to trust and believe that the vendor has conducted such a holistic review of its commercial software offerings. We have to trust the vendor, because there is little likelihood that any third party can attest to a vendor's approach to ridding its software of vulnerabilities. This specific point has been a thunderous charge of the OSS community, and we do not contest that assertion. But we caution that just because the software is open to review, it should not automatically follow that such a review has actually been performed (but of course you are more than welcome to conduct that review yourself—welcome to the bazaar).

Making Lightning Strike Twice

Instances of successful OSS products such as Linux, Apache, Perl, sendmail, and much of the software that makes up the backbone of today's Web are clear indications that successful OSS activities can strike often. But like with lightning, we can ask, "Is it possible to predict where the next strike will be?" or "Is it possible to make the next strike happen?"

Continuing with this analogy, we can answer these questions to the extent that science will permit. Like lightning, meteorologists can predict the likelihood of severe weather in a metro region given atmospheric conditions. For OSS, it may be harder to predict the likelihood of success for an OSS product or activity, but certain conditions appear to be key, specifically:

- It is a working product. Looking back at many of the products, especially Apache and Linux, none started in the community as a blank slate. Apache's genesis began with the end of the National Center for Supercomputing Applications (NCSA) Web server. Linus Torvalds released Linux version 0.01 to the community in September 1991. Just a product concept and design in the open-source community has a far less likely chance of success. A prototype, early conceptual product, or even a toy is needed to bootstrap the community's imagination and fervor.

▪ It has committed leaders. Likewise as important is a visionary or champion of the product to chart the direction of the development in a (relatively) forward-moving direction. Although innovation and product evolution are apt to come from any one of the hackers in the development community, at least one person is needed to be the arbiter of good taste with respect to the product's progress. This is seen easily in the Apache project (the Apache Foundation).
▪ It provides a general community service. This is perhaps the closest condition to the business model for commercial software. It is unlikely that a commercial firm will bring a product to market if there is no one in the marketplace who will want to purchase that product. In the open-source community, the same is also true. Raymond (2001) points out a couple of valuable lessons:
· "Every good work of software starts by scratching a developer's personal itch." (p. 32)
· "Release early, release often. And listen to your customers." (p. 39)
· "To solve an interesting problem, start by finding a problem that is interesting to you." (p. 61).

From these lessons, there is a theme that talks to the needs of the developers themselves (personal itch) and a community need (customers or consumers who need the product or service).

▪ It is technically cool. You are more likely to find an OSS device driver for a graphics card than an accounting package. Feller and Fitzgerald (2002) categorized many of the open-source projects in operation, noting that a high percentage of those were Internet applications (browsers, clients, servers), system and system-development applications (device drivers, code generators, compilers, and operating systems/kernels), and game and entertainment applications.
▪ Its developers are also its users. Perhaps the characteristic that is most indicative of a successful OSS project is that the developers themselves are also the users. Typically, this is a large difference between OSS and commercial software. In commercial software, users tend to convey their needs (i.e., requirements) to engineers who address those needs in the code and then send the software back to the users to use. A cycle ensues, with users conveying problems and the engineers fixing and returning the code. However in OSS, it is more typical that a skilled engineer would rather repair the problem in the software and report the problem along with the repair back to the community. The fact that OSS products are technically

cool explains why many of the most popular ones are used typically by the developer community on a day-to-day basis. (Not many software developers we know use accounting packages!)

This is not to say that any product or activity exhibiting these conditions will, in fact, be successful. But those products that are considered to be successful meet all of them.

This leads us to the next question: "Is it possible to make the next strike happen?" In lightning research, scientists use a technique called *rocket-and-wire technique* to coax lightning from the skies to the ground for research purposes. In that technique and under the optimum atmospheric conditions, a small rocket is launched trailing a ground wire to trigger lightning discharges (Uman 1997). For OSS, a comparable technique might involve creating conditions that are favorable to OSS development but may fail to instigate a discharge from the OSS community.

At this point, we abandon our lightning analogy and observe (and dare predict) that there will be other successful OSS products and activities in the coming years. Furthermore, we surmise that such products will exhibit the conditions discussed previously. Whether they happen by chance or by design is difficult to tell.

In Closing

We view OSS as a viable source of components from which to build systems. However, we are not saying that OSS should be chosen over other sources simply because the software is open source. Rather like COTS and CSS, OSS should be selected and evaluated on its merits. To that end, the SEI supports the recommendations of the PITAC subpanel on OSS to remove barriers and educate program managers and acquisition executives and allow OSS to compete on a level playing field with proprietary solutions (such as COTS or CSS) in government systems.

Adopters of OSS should not enter into the open-source realm blindly and should know the real benefits and pitfalls that come with OSS. Open source means that everyone can know the business logic encoded in the software that runs those systems, meaning that anyone is free to point out and potentially exploit the vulnerabilities with that logic—anyone could be the altruistic OSS developer or the cyber terrorist. Furthermore, having the source code is not necessarily the solution to all problems: without the wherewithal to analyze or perhaps even to modify the software, it makes no difference to have it in the first place.

It should not follow that OSS is high-quality software. Just as in the commercial marketplace, the bazaar contains very good software and very poor software. In this report, we have noted at least one commercial software vendor that has used its role in the OSS community as a marketing leverage point touting the "highest-quality software," when in fact it is no better (or worse) than commercial-grade counterparts. Caveat emptor (let the buyer beware); the product should be chosen based on the mission needs of the system and the needs of the users who will be the ultimate recipients.

Note

1. See the COTS-Based Systems (CBS) Initiative Web site at http://www.sei.cmu.edu/cbs.