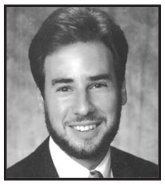

Steve McConnell

Open-Source Methodology: Ready for Prime Time?

Open-source software presents an approach that challenges traditional, closed-source approaches. Post your company's source code on the Internet for everyone to see? It seems crazy. But does the open-source approach work? No question about it. It already has worked on Linux, Apache, Perl, Sendmail, and other programs, and, according to open-source advocates, the approach continues to work marvelously. They will tell you that the software it produces is more reliable than closed-source programs, and defect fix times are remarkably short. Large companies such as Dell, IBM, Intel, Oracle, and SAP seem to agree. They have embraced open source's most famous program, Linux, and the Linux development community in particular sets an energetic example for the rest of the world to follow.

Considering that open source is an obvious success, the most interesting software engineering questions concern open source's future. Will the open-source development approach scale up to programs the size of Windows NT (currently at least four times as large as the largest estimate for Linux)? Can it be applied to horizontal-market desktop applications as effectively as it has been applied to systems programs? Should you use it for your vertical-market applications? Is it better than typical closed-source approaches? Is it better than the best closed-source approaches? After a little analysis, the answers will become clear.

The Source of Open Source's Methodology

Open-source software development creates many interesting legal and business issues, but in this column I'm going to focus on open source's software development methodology.

Methodologically, open source's best-known element is its use of extensive peer review and decentralized contributions to a code base. A key insight is that "given enough eyeballs, all bugs are shallow." The methodology is driven mainly by Linus Torvalds' example: Create a kernel of code yourself; make it available on the Internet for review; screen changes to the code base; and, when the code base becomes too big for one person to manage, delegate responsibility for major components to trusted lieutenants.

The open-source methodology hasn't been captured definitively in writing. The single best description is Eric Raymond's "The Cathedral and the Bazaar" paper, and that is sketchy at best (http://www.tuxedo.org/~esr/writings/cathedral-bazaar/cathedral-bazaar.html). The rest of open source's methodology resides primarily in the informal legend, myth, and lore surrounding specific projects like Linux.

Bug Me Now or Bug Me Later

In *Open Sources: Voices from the Open Source Revolution* (O'Reilly, 1999), Paul Vixie points out that open-source projects use extensive field testing and unmatched code-level peer review. According to Vixie, open-source projects typically have sketchy marketing requirements, no system-level design, little detailed design, virtually no design documentation, and no system-level testing. The emphasis on code-level peer review gives the typical open-source project a leg up on the average closed-source project, which uses little or no review. But considering

EDITOR-IN-CHIEF: Steve McConnell • Construx Software • software@construx.com

how ineffective the average project is, comparing open-source projects to the "average" closed-source project sets a pointless standard of comparison. Leading-edge organizations use a combination of practices that produce better quality, shorter schedules, and lower development costs than average, and software development effectiveness at that level makes a more useful comparison.

One of the bedrock realities of software development is that requirements and design defects cost far more to correct at coding or system testing time than they cost to correct upstream. The software industry has collected reams of data on this phenomenon: generally you can expect to spend from 10 to 100 times as much to correct an upstream defect downstream as you would spend to fix the same defect upstream. (It's a lot easier to change a line on a design diagram than it is to change a module interface and all the code that uses that module.) As Vixie points out, open source's methodology focuses on fixing all bugs at the source code level—in other words, downstream. Error by error, without upstream reviews, the open-source project will require more total effort to fix each design error downstream than the closed-source project will require to fix it upstream. This cost is not readily perceived because the downstream effort on an open-source project is spread across dozens or hundreds of geographically distributed people.

The implications of open source's code-and-fix approach might be more significant than they at first appear. By the time Linux came around, requirements and architecture defects had already been flushed out during the development of many previous generations of Unix. Linux should be commended for its reuse of existing designs and code, but most open-source projects won't have such mature, predefined requirements and architecture at their disposal. To those projects, not all requirements and architecture bugs will be shallow.

Open-source advocates claim that giving users the source code reduces the time needed for downstream defect correction—the person who first experiences the problem can also debug it. But they have not published any data to support their assertion that this approach reduces overall defect correction costs. For this open-source approach to work, large numbers of users have to be both interested in and capable of debugging source code (operating system code, if the system in question is Linux), and obviously doesn't scale beyond a small cadre of highly motivated programmers.

By largely ignoring upstream defect removal and emphasizing downstream defect correction, open source's methodology is a step backwards—back to Code and Fix instead of forward to more efficient, early defect detection and correction. This bodes poorly for open source's ability to scale to projects the size of Windows NT or to brand-new technologies on which insufficient upstream work can easily sink a project.

Not All Eyeballs Are Shallow

Open-source advocates emphasize the value of extensive peer review. Indeed, peer reviews have established themselves as one of the most useful practices in software engineering. Industry-leading inspection practices usually limit the number of reviewers to five or six, which is sufficient to produce software with close to zero defects on closed-source projects (Watts Humphrey, *Managing the Software Process*, Addison Wesley Longman, 1989). The question for open source is, How many reviewers is enough, and how many is too many? Open source's typical answer is, "Given enough eyeballs, all bugs are shallow." The more the merrier.

To most open-source projects, not all requirements and architecture bugs will be shallow.

About 1,200 programmers have contributed bug fixes and other code to Linux. What this means in practice is that if a bug is reported in Linux, a couple dozen programmers might begin looking for it, and many bugs are corrected within hours. From this, open-source advocates conclude that large numbers of reviewers lead to "efficient" development.

This answer confuses "fast" and "effective" with "efficient." To one of those people, the bug will turn out to be shallow. To the rest, it won't be shallow, but some people will spend time looking for it and trying to fix it nonetheless. That time isn't accounted for anywhere because many of those programmers are donating their time, and the paid programmers don't track their effort in any central location. Having several dozen people all looking for the same bug may indeed be fast and effective, but it is not efficient. Fast is having two dozen people look for a bug for one day for a total cost of 24 person-days. Efficient is having one person look for a bug eight hours a week for a month for a total cost of four person-days.

Economic Shell Game

A key question that will determine whether open source applies to development of more specialized applications (for example, vertical-market applications) is, Does the open-source methodology reduce development costs overall, or does it just push effort into dark economic corners where it's harder to see? Is it a better mousetrap or an economic shell game?

Considering open source's focus on downstream defect correction with significantly redundant peer reviews, for now the approach looks more like a shell game than a better mousetrap. It is appealing at first glance because so many people contribute effort that is free or unaccounted for. The results of this effort are much more visible than the effort itself. But when you add up the total effort contributed—both seen and unseen—open source's use of labor looks awfully inefficient.

Open source is most applicable when you need to trade efficiency for speed and efficacy. This makes it applicable to mass-distribution products like operating systems where development cost hardly matters and reliability is paramount. But it also suggests that open source will be less applicable for vertical-market applications where the reliability requirements are lower, profit margins are slim enough that development cost does matter, and it's impossible to find 1,200 people to volunteer their services in support of your application.

One-Hit Wonder or Formidable Force?

The open-source movement has not yet put its methodology under the open-source review process. The methodology is currently so loosely defined that it can hardly even be called a "methodology." At this time, the strength of the open-source approach arises largely from its massive code-level peer review, and little else. For open source to establish itself as a generalizable approach that applies to more than a handful of projects and that rises to the level of the most effective closed-source projects, it needs to fix four major problems:

- Create a central clearinghouse for the open-source methodology so it can be fully captured and evolved.
- Kick its addiction to Code and Fix.
- Focus on eliminating upstream defects earlier.
- Collect and publish data to support its claims about the effectiveness of the open-source development approach.

None of these weaknesses in open source's current development practices are fatal in principle, but if the methodology can't be evolved beyond its current kludgy practices, history will record open source's development approach as a one-hit wonder. If open source can focus the considerable energy at its disposal into defining and using more efficient development practices, it will be a formidable force indeed. ❖

Response

Open-Source Methods: Peering Through the Clutter

Terry Bollinger, Russell Nelson, Karsten M. Self, and Stephen J. Turnbull

Question: Have any of you ever experienced the following?

You are part of a new software development project. First, estimates are made of how much time it will take to develop the software. Next, the latest design tools and techniques are used to lay out the structure of the software. The completed graphical design, which fills several thick volumes, is handed over to a team of programmers who then begin coding furiously. After the coding effort is about 90 percent complete, the detailed metrics collected throughout the project indicate that it will be the most timely, wellplanned project ever completed by your organization. The integrators and testers then start putting all the code together—and discover to their surprise that the resulting system is just bit fodder for the next carnivorous Internet worm.

At that point, two or three experienced programmers take over all the real work. They use the bit fodder version as a sort of flabby prototype from which they can learn lessons about what *not* to do. After a couple of months of furious (and unscheduled) coding, they produce a passably working prototype—which management immediately ships to the customer as the first released version. Amazingly, the metrics previously attached to the bit fodder version are magically reassigned to the new prototype. Blame for not meeting schedule is assigned to the small team of experienced programmers who produced the prototype, of course, since they clearly behaved in an uncontrollable fashion.

The customer is not happy, but the first release (that is, the prototype) at least does something akin to what was ordered. The managers are not happy, but at least they have great metrics to prove how well they managed the project—as well as further proof of how headstrong programmers can mess up good schedules. The experienced programmers who created the working prototype are not very happy, either. They get most of their satisfaction from knowing that only *their* work produced anything useful.

Now, here is what open source really does: It gets rid of everything in the above story except for the last sentence.

In short, open-source methods cut though the clutter of overly hyped design methods, three-letter management fads, eye-of-the-newt metrics, brain-free programming, and managerial winking and nodding that are at the heart of so much of what sneaks by under the moniker of "software engineering." Open source simply demonstrates that there might be cleaner (and better) ways to do such things.

The Open-Source Razor

Is that to say that open source is some sort of total panacea for the future development of software? Of course not! Steve McConnell's essay aptly points out that, at present, open source scarcely even qualifies as a "methodology." It is more like a set of principles that define the absolute minimal process by which a large group of people can produce high-quality software. Such a minimalist methodology has its own merits, however. If nothing else, it acts as a sort of Occam's Razor for the rest of software engineering. Instead of asking, "How many more controls will this project need before it becomes predictable?" the Open-Source Razor demands that a new question be asked: "Can you justify adding a new control, method, or metric to the process when open-source methods already work fine without it?"

Open source demonstrates that there might be cleaner (and better) ways to do such things.

Efficiency, Efficiency… Who Has the Efficiency?

With all that said, let's take a look at this curiously minimalist methodology from another perspective: efficiency. Steve McConnell's essay makes the point that in traditional software development, the cost of finding and fixing a design defect increases dramatically as you move farther out into the life cycle of the software. Since open-source methods of inspection and bug fixing operate entirely at the source code level, doesn't that prove that open source will be vastly less efficient than traditional methods that catch defects as early as possible in the development process?

It's an excellent point, but it relies on the assumption that costs in open source work the same way as in traditional methodologies. This requires a closer examination of late-capture defect costs.

One such factor is that late fixes can affect large chunks of the overall software design, especially software that is not as modular as it should be. Another cost factor is the loss of the context of the original development effort—that is, the people, environment, and overall set of knowledge under which the software was originally developed. Reconstructing this environment can be very difficult after the project has concluded. Finally, it is much harder to figure out how erroneous behaviors correlate to specific code errors when the software is running in diverse locations with unique environments. Collectively, such factors make it a very good idea to catch design errors early in traditional closed-source development, before the context of the original development effort is lost.

The Open-Source Spiral Model

Open-source projects, however, attempt to ship out minimally working prototypes at the earliest possible time. By doing this they begin to receive feedback on their features and designs very early in the overall development process. It is this prototype-based feedback cycle that distinguishes open-source methods from a simple code-and-fix cycle. Indeed, open source is more accurately described as an unusually rapid and iterative form of Barry Boehm's famous spiral model of software development. Open source is not usually described in this way simply because the spirals—microspirals, actually—are wound so tightly together that to an outsider the entire effort looks much like a single large, "simple" coding effort. Inside, however, participants are vigorously iterating over requirements, design, coding, and testing activities, all going on within loops that might take as little as hours to complete. It is this highly iterative process that lies behind much of the reliability of open source, because each new fix can be vigorously rechecked in the subsequent loops of the microspiral.

Only open source has a plan for fixing bugs in the environment where they're discovered.

This process avoids the horror-story show-stopper bug that only shows up in shipped products. Only open source has a plan for fixing these bugs in the environment where they're discovered.

Bugs, Bugs… Who's Got the Bugs?

Another good point that Steve McConnell makes is that open source can lead to multiple people working on the same bug. However, there are two reasons why this effect of redundant bug fixes is probably not as important in practice as it might seem on paper.

First, it overlooks the complex personal network that develops naturally in open-source projects. Because the source code is fully available and contributions are fully attributed, participants are generally well aware of who the experts are for a particular type of bug. Mailing lists are also used to make these relationships even more explicit. The overall result is a natural deferring process whereby most participants immediately realize which subgroup or person is best suited to fix a particular type of bug.

Second, the "microcompetition" that occurs when multiple designers work on a single bug is not necessarily a bad thing in terms of overall efficiency of the open-source process. For example, poorly coded modules become like bait when microcompetition is possible. The weak modules attract many of the more skillful open-source developers in an effort to prove which one can come up with the best, most efficient, and longest-term fix. Especially when applied over time to many different modules, the result can be a very solid code base in which everyone saves time by not having to deal with the consequences of using slow, buggy modules.

Selective Evolution of Modularity

Open source also promotes efficiency by encouraging the evolution of high-quality modularity. This is a direct result of the decentralized nature of open-source development, in which only those source code modules that "make sense" to developers at remote sites can be efficiently updated by developers around the globe. Only source code that is rigorously modular, self-contained, and self-explanatory can meet such an objective. The overall result is that the decentralized organization of an open-source project is often mirrored in a finer structure of modules themselves, all the way down to a more rigorous standardization of their internal interfaces. While good closed-source organizations are of course aware of the benefits of good modularity, only open-source methods provide the kinds of individual incentives though which such practices can easily flourish and evolve over time. They also provide a warning about efforts such as Netscape's Mozilla that attempt to move weakly modularized proprietary code into open source. If the initial product is not already modular, such an effort is likely to fail before it really gets started.

Needed: Synergy!

We would like to end this response simply by applauding the type of synergy between open-source and traditional development that Steve McConnell is encouraging here. As demonstrated by the relationship between the Boehm spiral model and the open-source microspiral, there are important lessons for both sides. ❖

Terry Bollinger is an IEEE Software editor; in his spare time, he works at Mitre. Russell Nelson has made a living off free software since 1991. Karsten M. Self was happily programming SAS and hacking Unix environments when he got bit by the Linux bug in 1997; he now spends too much time thinking about how the free-software phenomenon works. Stephen Turnbull teaches economics in Japan and is professionally fascinated by the growth of open-source software, even compared with that of Japan. The authors can be reached via Nelson at nelson@crynwr.com.

IEEE SOFTWARE

Editorial: Send 2 electronic versions (1 word-processed and 1 postscript or PDF) of articles to Managing Editor, *IEEE Software*, 10662 Los Vaqueros Cir., PO Box 3014, Los Alamitos, CA 90720-1314; software@computer.org. Articles must be original and not exceed 5,400 words including figures and tables, which count for 200 words each. All submissions are subject to editing for clarity, style, and space. Unless otherwise stated, bylined articles and departments, as well as product and service descriptions, reflect the author's or firm's opinion. Inclusion in *IEEE Software* does not necessarily constitute endorsement by the IEEE or the IEEE Computer Society.

Circulation: *IEEE Software* (ISSN 0740-7459) is published bimonthly by the IEEE Computer Society. IEEE headquarters: Three Park Ave., 17th Floor, New York, NY 10016-5997. IEEE Computer Society Publications Office: 10662 Los Vaqueros Cir., PO Box 3014, Los Alamitos, CA 90720-1314; (714) 821-8380; fax (714) 821-4010. IEEE Computer Society headquarters: 1730 Massachusetts Ave. NW, Washington, DC 20036-1903. Annual electronic/paper/combo subscription rates for 1999: $27/34/44 in addition to any IEEE Computer Society dues, $49 in addition to any IEEE dues; $93 for members of other technical organizations. Nonmember subscription rates available on request. Back issues: $10 for members, $20 for nonmembers. This magazine is available on microfiche.

Postmaster: Send undelivered copies and address changes to Circulation Dept., *IEEE Software*, PO Box 3014, Los Alamitos, CA 90720-1314. Periodicals Postage Paid at New York, NY, and at additional mailing offices. Canadian GST #125634188. Canada Post Publications Mail Product (Canadian Distribution) Sales Agreement Number 0487805. Printed in the USA.