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 are directed toward 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 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 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.

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:

1. Create a central clearinghouse for the open-source methodology so it can be fully captured and evolved.

2. Kick its addiction to Code and Fix.

3. Focus on eliminating upstream defects earlier.

4. 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.

---

_Editor: Steve McConnell, Construx Software, 11820 Northup Way #E200, Bellevue, WA 98005._  
_E-mail: steve.mcconnell@construx.com - WWW: http://www.construx.com/stevemcc/_