Why OSS Folks Think SE Folks Are Clue-Impaired

Bart Massey
Computer Science Department
Portland State University
P.O. Box 751 M.S. CMPS
Portland, Oregon USA 97207–0751
bart@cs.pdx.edu

Abstract

The open source software development community has long been critical of mainstream software engineering thinking: mainstream software engineering has largely ignored or even scoffed at this critique. A summary of some key elements of such a critique can be useful in at least two ways: as part of an attempt to understand the existing relationship between SE and OSS, and as a tool for improving practice in both areas.

1 OSS Development Is Just SE... Right?

It is the perception of many members of the Software Engineering (SE) community that academic SE practice is the standard by which software development should be judged. In this view, high-quality scientific research on SE processes and tools drives high-quality software development practices that produce high-quality, low-cost software in almost any domain.

It is interesting to note that many experienced and capable software developers in commercial enterprises scoff at the notion of best SE practice as normative. This view is even more emphatic in the free and open source software (OSS) community. Reactions in response to the perceived mismatch between "real world" commercial and OSS development and SE "best practice" have run a gamut, from the development of customized and lightweight software development methodologies such as Extreme Programming [1] to the simple business-as-usual plan of ignoring the SE community altogether. Vixie [14] captures this attitude well, noting that "It is clear from historical examples that software need not be engineered to be widely used and enjoyed."

In fact, the ignorance of SE prescribed practice in the OSS community appears to be almost total and somewhat deliberate. This community has rejected the received wisdom of its peers and forefathers: OSS software is developed in a fashion that combines practices that would have been familiar 20 years ago (e.g., command-line debuggers, medium-level programming languages, implementation-centric construction) with idiosyncratic practices that change rapidly to reflect the OSS perception of best practice (e.g., distributed development and revision control tools, special documentation formatters and styles, use of novel communication channels such as IRC and Slashdot).

The details of this perceived mismatch between the OSS and SE communities is interesting on a couple of levels. First, the best folks in these communities are extremely smart, experienced people: their perceptions almost always contain at least a grain of truth. Second, the practices that each side conceives of as strengths are potential areas for improvement on the other side.

Two disclaimers. First, the focus in this paper is on informing the SE community of the perceptions and insights of the OSS community. The obvious companion piece can and should be written, entitled something like "Why the SE Folks Think The OSS Folks Are Disfunctional." In an SE venue, that would be preaching to the choir—in an OSS venue, it could be useful. Second, please note that this paper is not entitled "Why The Author Thinks SE Folks Are Clue-Impaired." The author has nothing but the deepest respect for both sides of this dialog: indeed, has as an important goal the deepening of understanding and inevitably respect between these communities. However, it is likely that this goal can be achieved only when each side fairly and accurately understands the other's position. Larry McVoy (as quoted in [6]) has observed that "it is far better to figure out a way to allow the business world and the hacker world to coexist and benefit one another."

The remainder of this paper proceeds as follows: The relationship between OSS and SE development processes and practices is explored, concentrating on the areas where OSS

finds SE lacking. A review of outstanding issues suggests shared engineering ground for the OSS and SE communities. Finally, after summarizing the situation, directions are suggested for further progress.

2 The Cathedral

The very title of Eric Raymond's essay *The Cathedral And The Bazaar* [8] contains an implicit critique of the canonical practice of software engineering. The content accurately reflects this title: Raymond suggests that the commercial SE community suffers from the sort of insularity, absolutism, and cloistered mentality often attributed to the priesthood of the Middle Ages. This perception is especially significant in that Raymond's essay is widely cited and admired within the OSS development community. While the focus of the essay is on the Bazaar, the image of the Cathedral is an iconic summary of the belief of even some of the most sophisticated members of the OSS development community.

The detailed criticisms of "academic" SE by the OSS community fall into two general categories. First, the prescriptive and descriptive process models of software development apostolized by the SE community are widely viewed by the OSS community as inappropriate for their work. Second, many of the specific techniques and methods of the lay software development practitioner, as endorsed by the SE community, are seen as anathema in the OSS community.

2.1 Processes

There is a widespread belief among OSS developers that the SE community is insufficiently driven by actual experience in software development. In this view, ritual replaces sense, and the resulting ignorance and superstition damages software development.

Some of the specific process models and prescriptions detailed here are common in industry, some are less so. All are blessed by at least a significant portion of the SE community. It is worth recalling the point made earlier: the critique detailed here is useful inasmuch as it helps to inform the SE community of perceptions, and to illuminate areas that need work.

The Waterfall Model The Waterfall Model [10] of the software development process in its pure form is largely out of favor even in the modern SE community. Nonetheless, it is still often taught in introductory coursework, and it is widely and perhaps correctly perceived as reflecting and underlying most modern SE process models. As taught by the author and his colleagues in the Oregon Master of Software Engineering program, this model breaks the software development process into six stages: User Requirements Elicitation; Requirements Specification; Architectural Design; Detailed Design and Implementation; Validation and Verification; and Maintenance.

The OSS community that the Waterfall Model arose from the need to monitor contract compliance in extremely large U.S. Department of Defense software projects in the 1960s and 1970s. The question then arises naturally: how can such a model be relevant to the development of small, open volunteer projects in the 21^{st} century?

The author tends to explain the Waterfall Model in the classroom and in discussions with OSS developers in terms of a similar and more common-sense structure, consisting of "what to build", "how to build it", "build it", and "is it built right?" stages. This doesn't seem to help much. The fundamental perceptual mismatch can be captured by such OSS slogans as "code is cheap" and "the code is the documentation". The belief of much of the OSS community is that the "build it" phase can expand to encompass all of the other phases. The SE community, of course, tends to consider this experiment as tried and failed: it is precisely the disasters of implementation as engineering in the commercial world that led to the Waterfall Model in the first place.

The response of the OSS community to the past failures of implementation-centric engineering is to distinguish themselves in several ways. First, they argue that technological improvements in tools in the past 20+ years, particularly in high-level programming languages and communications infrastructure (ala the Internet) have permitted a qualitative change in the way programmers can collaborate successfully. Brooks' [2] chapter on documentation is the prototypical argument here: while much of the rest of this book is timeless, that chapter contains much advice that is simply archaic in present-day programming reality.

Second, they argue that the Waterfall Model and its many descendants are an inevitable result of the contract model of software procurement. In this view, OSS requirements can be "Make the users and the programmers happy," design can be "Do something cool," and V&V can be "Does the software do something cool and make the users and the programmers happy?" Vixie notes that "Open Source folks tend to build the tools they need or wish they had. . . . [Later] other users will start to either ask for features or just sit down and implement them and send them in." Given this premise, much of the need for formalized process of any kind goes away.

Requirements Engineering The author has written previously about the sources of OSS requirements [4]: in the interests of brevity, that discussion will not be repeated here. The critique of the SE requirements process, on the other hand, is worth making explicit. The process of system-

atically gathering and correctly formalizing requirements requires tremendous effort. To be worthwhile, this effort must pay for itself. Commercial product requirements efforts tend to pay for themselves in one of several ways, none of which seem to apply in great detail in OSS projects.

For example, in OSS there is little money to be made by correctly gauging the market. While OSS developers do actively seek a large user base, there appears to be little concern in most OSS projects about squeezing the last few percent of the market out. In addition, competition is viewed almost entirely differently in this altruistic marketplace: it is quite common for projects competing for "customers" to be nonetheless cooperating to improve the quality of each others' projects. In this environment, market requirements can be determined by watching the users of competing projects to see what features and –ilities they are actively seeking, and then getting the assistance of the competition in providing them.

As another example, the need to correctly determine requirements for safety-critical and mission-critical systems appears to be less of a concern in the OSS community. Highly critical systems are not a domain in which much OSS software is fielded. Systems that are fielded in, for example, security-critical situations tend to have requirements that are of one of two kinds: either extremely simple, or mandated by some (usually well-crafted) existing standard. Neither case demands much in the way of systematicity or even care in requirements elicitation or specification.

As a result of these kinds of factors, it is difficult to tell an open source developer that it is a good idea to spend 30% or more of the product development cycle on requirements. First of all, as discussed below, the whole notion of a product development cycle is really rather foreign to OSS. Second of all, the OSS developer will almost always perceive this advice as totally out of tune with the realities of their project.

Design Engineering The traditional SE view of design involves a great deal of design effort, usually in some combination of top-down (preferred) and bottom-up approaches, and culminating in a meet-in-the-middle step. The OSS view is the exact opposite: a small amount of middle-out design concentrated on a layer somewhere just below the top, followed by extensive design-by-coding.

Is it possible to get a quality product out of this design process? The OSS folks think so. Linus Torvalds [12] writes that "Linux has succeeded not because the original goal was to make it widely portable and widely available, but because it was based on good design principles and a good development model." The keys, in this view: extraordinarily sharp modularity, with the narrowest possible couplings and broadest possible coherencies; the "code is cheap" philosophy of treating the entire implementation as a partially-reusable design prototype; and the use of high-level scripting languages and sophisticated environments to produce usable implementations via rapid development.

It must be noted that successful medium-sized and large OSS projects tend to be architected by developers of extraordinary skill and experience. This tendency is balanced, however, by the express desire and demonstrated ability to substitute clusters of small, largely decoupled projects for the medium-sized and large ones. In the case of successful but necessarily large OSS applications, the underlying architecture and implementation often begins as an inheritance from a traditional SE project, as in the case of the X Window System, Mozilla, and OpenOffice.org.

Structured Testing The principal problem with structured testing from an OSS point of view is similar to that of sophisticated requirements or design engineering: it is quite demanding of resources. Worse yet, as in the commercial world, traditional structured testing activities are viewed in the OSS world as repetitive and unrewarding for the practitioner. While professional testers receive compensation in the form of salary, the reward system for systematic testing among OSS developers is practically nonexistent.

Further, even traditional software development models have taken the view that "user testing" on real workloads is a powerful testing tool for reducing the expected pain from software failures. A comparison with Mills' Cleanroom [5] approach is appropriate: in OSS, the formal development portion of Cleanroom is replaced with the "many eyes make shallow bugs" approach to generating low-defect software for the user testing phase.

In OSS, as with much COTS software development, user testing takes the form of fielding new software as early as possible, and waiting for the users to report failures. In OSS, this process is supported by the continuous coexistence of deployed experimental and stable versions, and by the fact that much of the user base itself consists of professional-quality developers that can understand the application source and provide accurate failure reports, do their own defect analysis, and suggest solutions. DiBona et. al. [3] write that "By sharing source code, Open Source developers make software more robust. Programs get used and tested in a wider variety of cntexts than one programmer could generate, and bugs get uncovered that otherwise would not be found. Because source code is provided, bugs can often be removed, not just discovered, by someone who otherwise would be outside the development process."

Software Maintenance The maintenance phase of traditional SE is often acknowledged as a primary part of the programming effort, and yet little has been prescribed or even described about software maintenance. The defined process maintenance acknowledged by the SE community

in the past largely centered around defect removal; while this is changing over time, the central topics in maintenance still tend to be subjects such as CASE tools and Change Control Boards that are of limited direct relevance to the OSS community.

Indeed, most of the maintenance process commonly employed in the OSS community is integrated at a fine grain into the development process. The use of software version control tools and defect tracking and reporting tools is ubiquitous, and typically begins early in the (already foreshortened) OSS development cycle.

A final important factor in OSS maintenance, as with testing, is the amount of activity that is user-centered. OSS projects rarely suffer from lack of maintenance: the users are empowered and encouraged to undertake maintenance activities on their own behalf. The overall effect of this OSS maintenance process is to integrate maintenance into the system life cycle in a way that systems developed using traditional SE maintenance models have difficulty approximating.

2.2 Practices

The processes and process models espoused by SE are one thing. The practices of commercial software development are quite another. It is worth considering several of the practices on which the SE community looks most askance.

Episodicity The software development cycle is often described as a "life cycle". This metaphor is odd in a couple of ways. First, it is not clear what the "rebirth" part of the software life cycle is: while there are some obvious candidates, none is a perfect fit. More importantly, the metaphor describes a development cycle that is episodic, moving through a series of distinct phases on its way to completion.

The OSS development process is conceptualized as much more of a steady-state affair. From quite early on in an OSS project, developers integrate design of new features, release of new code, and defect removal and maintenance activities in a highly integrated fashion. While the larger projects might have, for example, feature freezes, these are viewed as necessary but anomalous interruptions in this process of continuous development and dissemination. "Release early and often" is the adage that goes with this practice in the OSS community: there is a widespread belief among OSS developers that continuous improvement, not of processes but of codebases, is a more productive approach than the episodic approaches of the commercial world.

Obscurantism The nature of OSS projects is to be transparent. Not only is the code freely available for study, but any available documentation and insight about the system is freely shared, and the developers themselves often commit significant amounts of time to assisting with and explaining the software. One of the principal frustrations with the commercial software development world is the opposing attitude that users[1] should expect to understand the absolute minimum needed to operate an opaque system.

The SE insistence on detailed user requirements plays into this attitude of secrecy in a subtle way. The attitude taken by commercial developers seems to be that any program that conforms to the specification should be sufficient for the user's needs. Further, the user specification often becomes embodied as a user manual: these often opaque or defective manuals are considered to meet the requirement of software description.

The logical conclusion of this sort of philosophy is in the area of cryptography and security. OSS developers generally insist on open implementations following Kirchoff's Principle [9]: the only secret is the key itself. Commercial vendors tend to build software systems where the key, the algorithm, the implementation, and the protocols are obscured, and the user interface itself is somewhat so.

Finally, the law is used to obscure the understanding of software. Michael Tiemann [11] writes "Outside [the walls of proprietary software companies], the use and distribution of that software is heavily controlled by license agreements, patents, and trade secrets. One can only wonder what power, what efficiency is lost by practicing freedom at the micro level and not at the macro level."

Complexification Larger OSS systems are quite complex. Nonetheless, there is a serious drive in the OSS community to keep systems small, simple, and separate. This is perhaps partly a reaction to the problems the Waterfall Model was trying to tackle. The alternative to simplicity is careful management of complexity. In the opinion of most in the OSS community, complexity should be tolerated only when it cannot be avoided. Einstein's "as simple as possible but not simpler" is the watchword here. Torvalds [13] writes that "You should absolutely not dismiss simplicity for something easy. It takes *design* and good taste to be simple."

It is interesting to note that some of the largest and most complex OSS systems in widespread deployment have at least some roots in traditional SE development. As noted previously, Mozilla and OpenOffice.org both qualify in this regard. Other large and complex open source systems, such as the KDE and Gnome desktop environments, are structured in similar ways to the older systems whose style they generally imitate. More OSS-style projects, such as Apache, tend to be more modular, with simpler pieces and less glue. Even the Linux kernel, while of necessity highly

[1] In this context, OSS folks spell this word with a leading 'l' [7]

complex, is understood by most kernel developers as a set of highly isolated and reasonably simple components connected by narrow interfaces. This philosophy is often referred to as the "UNIX way", after the system that inspired it: indeed, the Multics system that was an inspiration to the developers of the original UNIX was an extremely complex, monolithic system built using traditional SE practices.

3 A Clue Stick For Everyone

This paper started out with a question: "OSS development is just SE... right?" Apparently, the OSS community would beg to differ. The issues discussed above are just some of those that OSS developers and users discuss every day in fora such as Slashdot.org.

The question that remains is simple to state, but difficult to answer: is the OSS critique accurate? Is academic/industrial SE a glass cathedral, whose fragility is exceeded only by the danger to those who work within it? Is it necessary to move to new models of SE to make progress in software development? Or are the OSS critics self-deluded advocates of a methodology that will eventually fall prey to the same problems that have befallen other attempts to do distributed, informal development of weakly specified, designed, and tested software?

The author must answer with a predictable and resounding "maybe." Certain SE practices have long been regarded by the SE community itself as questionable. The pure Waterfall Model is largely dead, although its descendants in the community of episodic models live on. The disconnect between commercial software development and theoretically good SE practice has long been noted, although it is not clear whether to close this gap via "improved" practices, "improved" theory, or some combination of the two. The maintenance phase of software development needs to be better understood, and more strongly and flexibly supported by SE methodologies. The emphasis on large, tightly-coupled systems in SE needs to be decreased.

At the same time, the OSS community needs to understand that they are only now beginning to encounter many of the problems that led to the serious study of SE. As software systems grow, they become more unreliable and more difficult to manage at an alarming rate. In domains such as safety-critical and mission-critical systems, informal approaches are not enough. Finally, as the userbase for OSS becomes larger and the concentration of sophisticated users and developers is diluted, it is important that the OSS community evolve software methodologies that are compatible both with the tenets of the open source revolution and the needs of the user community.

Acknowledgments

In a paper of this sort, it is not clear that the author is doing anyone a favor by acknowledging their contribution by name. Nonetheless, it is worth acknowledging a few of the folks that arguably can take it. In addition to being close personal friends, Keith Packard and Mike Haertel are two top OSS figures with whom the author has had frequent conversations on the role of SE in OSS development: Keith also provided much useful commentary and advice on this paper. On the SE side, Warren Harrison has been a terrific sounding board for ideas and a fruitful source of suggestions. Finally, the author would like to thank the participants in the Second ICSE Workshop on Open Source Software Engineering for providing the inspiration for this essay.

References

[1] K. Beck. *Extreme Programming Explained*. Addison-Wesley, 2000.

[2] F. Brooks. *The Mythical Man-Month: 20th Anniversary Edition*. Addison-Wesley, 1995.

[3] C. DiBona, S. Ockman, and M. Stone, editors. *Open Sources: Voices of the Open Source Revolution*. O'Reilly, 1999.

[4] B. Massey. Where do open source requirements come from (and what should we do about it)? In *Proc. 2nd Workshop On Open-Source Software Engineering*, Orlando, FL, May 2002.

[5] H. D. Mills, M. Dyer, and R. Linger. Cleanroom software engineering. *IEEE Software*, 4(5):19–25, Sept. 1987.

[6] G. Moody. *Rebel Code: Inside Linux and the Open Source Revolution*. Perseus Publishing, 2001.

[7] E. Raymond and G. L. Steele, editors. *The New Hacker's Dictionary*. MIT Press, 1991.

[8] E. S. Raymond. *The Cathedral & the Bazaar*. O'Reilly, 2001.

[9] B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. Wiley, 1994.

[10] I. Sommerville. *Software Engineering*. Addison-Wesley, sixth edition, 2000.

[11] M. Tiemann. Future of Cygnus Solutions: A entrepreneur's account. In DiBona et al. [3], pages 71–89.

[12] L. Torvalds. The linux edge. In DiBona et al. [3], pages 101–111.

[13] L. Torvalds and D. Diamond. *Just For Fun: The Story of an Accidental Revolutionary*. HarperBusiness, 2001.

[14] P. Vixie. Software engineering. In DiBona et al. [3], pages 91–100.