High Quality and Open Source Software Practices

(Position Paper)

T. J. Halloran
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
thallora@cs.cmu.edu

William L. Scherlis
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
wls@cs.cmu.edu

1. INTRODUCTION

Surveys suggest that, according to various metrics, the quality and dependability of today's open source software is roughly on par with commercial and government developed software. What are the prospects for advancing to much higher levels of quality in open source software? More specifically, what attributes must be possessed by quality-related interventions for them to be feasibly adoptable in open source practice? In order to identify some of these attributes, we conducted a preliminary survey of the quality practices of a number of successful open source projects. We focus, in particular, on attributes related to adoptability by the open source practitioner community.

2. OBSERVED QUALITY PRACTICE

We conducted a preliminary survey of a number of open source projects to gain a clearer understanding of the practices used by active and successful open source projects. We examined the publicly visible portions of these projects from November 2001 through March 2002, with a focus on technical approaches to collaboration and quality. A summary of results for eleven of the projects surveyed is presented in Table 1. The SLOC[1] counts for the predominate languages are shown in the table in order to provide a rough gauge of code size and complexity. The projects all utilized web portals to encapsulate collaboration tools, and these tools are inventoried in the table. We also note unusual project-specific quality practices beyond personnel practices (i.e., limiting commit privileges) and the obvious elements of "micro-process" (particularly the ubiquitous nightly build).

The surveyed projects focus on *people* as the locus of Quality Assurance (QA). Clearly, responsibility for quality rests on those who have code commit privileges. Any code received from a programmer without commit privileges (usually attached to a bug report in patch format) must be reviewed and accepted by a programmer with commit privileges. In addition, more elaborate processes such as Mozilla's *review* and *super review* and NetBeans' *high resistance* ensure that lead programmers review code changes.

[1] We used the SLOCcount program by David A. Wheeler.

All projects use nightly builds. The nightly build ensures the source code still compiles after the day's changes. Most projects include some regression testing in the build process, and some projects also generate daily binary builds for public download. The Mozilla and Perl projects use the *Tinderbox* tool to help assure portability by building the code on multiple platforms after *any* code change; if the build breaks on any platform, then all subsequent code changes are limited to fixes.

A publicly visible bug and issue tracking tool is used by nearly all the projects we examined (one of the projects allows bug submission but not public status viewing). Users post bugs and enhancement requests. Each such post becomes, in effect, a tiny public mailing list focused solely on that issue. Some of the discussions are resolved rapidly (e.g., "invalid; not a bug") while others can last for weeks and include tens of messages. The bug and issue tracking tools provide the vehicle for contributions of source code from programmers without source code commit privileges. In addition, the issue tracking tool has a role in project management and tracking, enabling, for example, specific issues (performance, functionality, bug repairs) to be linked to a "stable release issue," thus setting a threshold for a subsequent release.

Because the developers in most of the successful projects are also users, the distinction between "bug report" and "feature request" in many of the issue reports is often not a sharp one. Receiving duplicate or invalid issue reports is common and specific bug responsibility can sometimes bounce between several programmers (or groups) before finally being accepted.

3. QUALITY INTERVENTION CRITERIA

Informed in part by this survey, we suggest below some key attributes of potential quality interventions focused on open source.

3.1 Process and the Walled Project Server

There is an important distinction in open source projects between the policies of visibility (including both access management and intellectual property disposition of the shared software assets) and the policies of engineering practice (including particularly architecture, tool use, process, and roles of people). The fact that open source projects, in theory, allow anyone to view and copy the source code (or even start a competing open source project) raises an interesting question: What do the leaders of an open source project

Table 1: Open Source Project Size, Collaboration Tool Use, and Quality Practice Summary

Name	Size & Composition kSLOC	Language	Collaboration Tools Used	Notable Quality Practices
Apache HTTPD Server	77	*Total*	CVS, Ezmlm, GNATS, ViewCVS	• Proposed changes are voted upon via mailing list; 3 yes votes with no vetos are required to commit a change. • Bugs are reported via the GNATS bug tracking tool unless they expose security problems in which case a private mailing list is encouraged.
	70	C (92%)		
	5	sh (7%)		
GNOME	1,338	*Total*	CVS, Bonsai, Bugzilla, IRC, LXR, Mailman	• Efforts to start a volunteer GNOME QA team are underway but today GNOME testing is done by individuals who report any bugs they find via the Bugzilla issue tracking tool. • GNOME has documented programming guidelines.
	1,207	C (90%)		
	86	sh (6%)		
GNU Compiler Collection (gcc)	1,182	*Total*	CVS, DejaGnu, Ezmlm, GNATS, MHonArc	• Uses a very detailed test suite built around the DejaGnu testing tool. • The project requires use of the GNU coding standards plus some gcc specific coding standards. • A successful conversion from Cathedral to Bazaar development.
	884	C (75%)		
	163	C++ (14%)		
	65	Java (6%)		
Jakarta Tomcat	59	*Total*	CVS, Ezmlm, Bugzilla, ViewCVS	• The Sun Java coding conventions are required for all Java code. • Roles within the Jakarta project are well defined and a documented process is used to resolve issues within the project.
	44	Java (75%)		
	14	C (24%)		
KDE	2,050	*Total*	CVS, Code/CVS Web, DebBugs, LXR, Mailman	• A "KDE Core Team" uses a democratic voting procedure to resolve important issues. They set KDE's overall direction and release schedule. • KDE developers conform to several industry and *de facto* standards.
	1,717	C++ (84%)		
	228	C (11%)		
Linux Kernel	2,570	*Total*	CVS, CVSWeb, LXR, Mailman, Majordomo	• Roughly 50 kernel mailing lists focus kernel work. • Bugs and quality issues may be discussed on the kernel mailing lists or reported via a specific Linux distribution's web portal.
	2,415	C (94%)		
	146	asm (6%)		
Mozilla	2,170	*Total*	CVS, Bonsai, Bugzilla, IRC, LXR, Tinderbox	• Developers use an evolving C++ portability guide. • A *review* and *super-review* process is required for most code changes. Potential patches are submitted to Bugzilla as attachments for review. • An active volunteer QA group helps check WC3 and IETF conformance. • Strong portability testing monitored by Tinderbox.
	1,358	C++ (62%)		
	749	C (35%)		
NetBeans	758	*Total*	CVS, Bugzilla, ViewCVS	• A strong QA group exists (supported by Sun). *High resistance* process (bug justification/approval & code review) used before stable releases. • NetBeans uses SourceCast web portal by CollabNet (strong integration).
	753	Java (99%)		
Perl	313	*Total*	CVS, MHonArc, Perlbug, Request Tracker, Tinderbox	• Strong portability testing monitored by Tinderbox. • Perlbug being used for user Perl5 bugs, Request Tracker is being used for Perl6/Parrot issues. • CPAN Internet module installation includes module testing.
	161	Perl (51%)		
	123	C (39%)		
Python	384	*Total*	CVS, CVSWeb, Mailman	• Python uses SourceForge.net web portal by OSDN (strong integration). • SourceForge patch manager nicely organizes community contributions to core developers (40 core Python developers).
	192	Python (50%)		
	185	C (48%)		
XFree86	1,943	*Total*	CVS, Mailman, ViewCVS	• No public bug tracking system utilized, a simple HTML bug form only. • To become an XFree86 developer you must submit an accepted patch and ask to join—you are then allowed to join developer mailing lists.
	1,833	C (94%)		
	36	C++ (2%)		

actually control? In a nutshell, they control the composition, configuration, and information flow in and out of their project's server. And in this way they can exercise tight control over the engineering practices of the project—not by limiting the behavior of individual developers in their own personal space, but by limiting the kinds of transactions developers can make with the persistent project state on the server.

Metaphorically, the project server is surrounded by a *wall*. This wall, embodied by the web portal and open source collaboration tools, is designed to *maximize* outgoing information flow and simultaneously to *strictly limit and control* incoming information flow. The critical processes of the project are enacted behind the server wall. These processes include code commits, documentation management, configuration management, builds, regression tests, and the like.

The walled project server thus physically embodies critical aspects of software best practice (principally process, build management, design record, and architecture) as defined by the project leaders. This enables effective engineering management despite the fact that the engineers are self-selected and largely self-managed. This enables combining managed rigor in project-level practice with a complete lack of restraint on the client-side developer space. Any quality-related intervention must respect this overall approach.

3.2 Communication

A ubiquitous, almost defining, trait of open source practice is that *tool mediation is the norm*. This enables leaders to shift the burden of policy enforcement from people to tools. Tools support authentication, regulation of commit privileges, audit and notification, and other policy-related functions.

One of the reasons why tool mediation is possible to this (relatively) unusual extent is that developers are rarely co-located. A consequence is that developer communication is almost always mediated through the project server, creating an *ad libitum* organizational memory. Indeed, projects with significant co-located groups, such as Mozilla and NetBeans, must take explicit measures to reinforce the social norm of computer-mediated developer communication. The textual organizational memory is semi-structured, in the sense that the discussions are situated in topic- or issue-specific mailing lists. While the conversation is focused on issue resolution, it also (at negligible marginal cost) records rationale for later use. Indeed, transforming this "collective stream of consciousness" into more structured design documents is a social engineering challenge for most open source projects.

3.3 Boot-Strapping

Table 2 categorizes the collaboration tools reported in Table 1. With the notable exception of the SourceCast web portal, all the tools listed are open source. It is often observed that many open-source projects engage in *boot-strapping*—the use of open source tools on open source projects. Indeed, we find boot-strapping common for collaboration tools.

While boot-strapping is often rationalized on the basis of ideology, in fact it has two significant effects. First, it lowers the barrier to entry for new participants in an open source project, enabling participants to create their personal (client-side) developer sites at low cost and without reliance on particular products. Second, it enables developers to fluidly shift their attention from tool use to tool development and repair. This movement up and down the tool food chain reinforces the developers-as-users principle.

A seeming exception to the principle of "tool openness" is the use of CollabNet's proprietary SourceCast web portal by projects such as NetBeans. This web portal does not raise an entry barrier, however, because it is built around familar open source tools. In addition, the proprietary portion of SourceCast exists only on the project server—the tools used by project participants on their individual computers remain open source or at least cost-free. This may be a significant distinction. It also raises the question of whether ideology is the driver or perhaps just a *post hoc* explanation for many successful open source practices. Nonetheless, it is clear that ideology within many open source projects today means that only open source tools will be adopted.

3.4 Incrementality and Gentle Slope

Most project portals offer resources to help potential new participants quickly reach the point of becoming visible and acknowledged contributors to the project. We can hypothesize that this desire to create a "gentle slope" learning curve [this term due to Michael Dertouzos] has yielded a *de facto* consensus on infrastructural tools such as Apache, CVS, and others. It also increases the payoff for developers down-shifting in the food chain from tool user to tool developers.

In addition to tool selection, "gentle slope" is reinforced by the communication practices noted above, in which increments of contribution can be made by participants, enabling other participants to build on the results and providing an explicit and objective record of the personal role of the contributor.

To the extent that it can be achieved, the incremental "gentle slope" model has significant effects for a project and its participants. It enables certain categories of participants to gracefully engage and disengage in a project without adverse consequences to the project. The entry model for many projects reinforces this—with new participants encouraged to review, debug, and document existing code, contributing changes in the form of mailing list posts for the attention of those with commit privileges. The *de facto* tool consensus reduces the barrier for developer-side tooling and for acquiring skills to interact effectively with the project server. As the new participant begins to do more work with the source code, such as private builds, additional increments of tool investment are required. The point is that new participants do not incur significant risk with respect to their time and resources in order to engage, perhaps experimentally, in collaborating on a project.

Table 2: Open Source Collaboration Tools

Tool Category	Name (Table 1 occurrence frequency)
Web Portal	Custom(9), SourceForge.net(1), SourceCast(1)
Source Code Control	CVS(11)
Code Viewers	ViewCVS(4), LXR(4), CVSWeb(3), Bonsai(2), CodeWeb(1)
Mailing List Management/Archive	Mailman(5), Ezmlm(3), MHonArc(2), Majordomo(1)
Bug/Issue Tracking Tools	Bugzilla(4), GNATS(2), DebBugs(1), Perlbug(1), RequestTracker(1)
Test Support Tools	Tinderbox(2), DejaGnu(1)
Instant Messaging	IRC(2)

3.5 Evolutionary Focus

The projects listed in Table 1 all work within what would conventionally be regarded as software maintenance and evolution phases of the software lifecycle. Rarely, if ever, would any of these projects create a new version of the software without reuse of an existing code base. Indeed, it can be hypothesized that there is great sensitivity in open source practice to architecture design. The longstanding success of gcc and Linux, for example, derives in many ways from architectural prescience in the initial designs—enabling both successful division of effort on the part of develop teams and long-term incremental growth in capability without excessive quality compromise.

4. QUALITY PROSPECTS

What characteristics must a quality-related technology or practice possess in order to be feasibly adoptable in projects following open source practices? Based on the observations above, we suggest some likely criteria: (1) an incremental model for quality investment and payoff (e.g., incrementally adding analysis support, test cases, measurement, or other kinds of evidence collection), (2) incremental adoptability of methods and tools both within the server wall and in the baseline client-side tool set, (3) a trusted server-side implementation that can accept untrusted client-side input, and (4) a tool interaction style that is adoptable by practicing open source programmers (i.e., that does not require mastery of a large number of unfamiliar concepts).

With the exception of testing technology and some code analysis technology, these requirements suggest that some adaptation will be required before adoption is possible for tools that embody, say, lightweight formal methods approaches or advanced program analysis approaches. Clearly, any technique or tool is not feasibly adoptable if it requires a major (client-visible) overhaul of a project web portal, collaboration tools, development tools, or source code base. Discernible increments of benefit from increments of participant effort is key to adoptability.