Leveraging Open-Source Communities To Improve the Quality & Performance of Open-Source Software

Douglas C. Schmidt
University of California, Irvine
Electrical & Computer Engineering Dept.
schmidt@uci.edu

Adam Porter
University of Maryland
Computer Science Dept.
aporter@cs.umd.edu

This position paper was submitted to the First Workshop on Open-Source Software Engineering, 23rd International Conference on Software Engineering, Toronto, Canada, May 15, 2001.

Abstract

Open-source processes have emerged as an effective approach to reduce cycle-time and decrease development and quality assurance costs for certain types of software. they are not without challenges, however, such as decreasing long-term maintenance and evolution costs, improving quality assurance, sustaining end-user confidence and good will, and ensuring the coherency of system-wide software and usability properties. Although aspects of these issues are unique to open-source development, well-organized open-source projects make it easier to address certain of these challenges compared with traditional closed-source approaches to building software.

We have begun a long-term research effort, called Skoll, whose goal is to leverage common open-source project assets, such as their technologically sophisticated worldwide user communities, to devise techniques that address key challenges of open-source software development. In particular, we are conducting a long-term case study of two widely used open-source projects, ACE and TAO, to design, deploy, and evaluate techniques for improving quality through continuous distributed testing and profiling. This position paper presents our view of the pros and cons of open-source processes and outlines the work we are doing to improve the quality and performance of open-source software.

Introduction: Why Open-Source Works

Over the past decade, open-source processes have emerged as an effective approach to reduce cycle-time and decrease development and quality assurance (QA) costs for certain types of software, such as:

- Operating systems, e.g., Linux, FreeBSD, and NetBSD
- Web servers, e.g., Apache and JAWS
- Middleware, e.g., ACE+TAO, MICO, omniORB, and JacORB
- Compilers, e.g., GNU C/C++/Ada
- Editors, e.g., GNU emacs
- Language processing tools, e.g., Perl, GAWK, Flex, and Bison
- System/network support tools, e.g., Samba, Sendmail, Bind

These open-source projects are distributed under a variety of licensing schemes[1] and business models. In general, however, they all distribute their software in source code form, allow derived works to be created and freely distributed, and rely upon their user communities for many development and QA activities traditionally performed internally by software vendors.

From an end-user perspective, it is our position that open-source has succeeded for the following reasons:

- *It reduces the cost of software acquisition*. Open-source software is often distributed without development or run-time license fees. Moreover, open-source projects typically use low-cost distribution channels, such as the Internet, to allow rapid access to source code, examples, regression tests, and development information.
- *It leverages and enables diversity and scale*. Well-written and well-configured open-source software generally ports easily to a variety of heterogeneous OS and compiler platforms. In addition, open-source provides end-users with the freedom to modify and adapt their source base rapidly to meet new market opportunities.
- *It simplifies collaboration*. Open-source promotes the sharing of programs and ideas among members of technology communities that have similar needs, but whom also have diverse technology acquisition/funding strategies.

From a software process and productivity perspective, it is our position that open-source projects are successful for the following reasons:

- *Leveraging the user community effectively*. In today's time-to-market-driven economy, few software providers can afford long QA cycles. As a result, nearly everyone who uses a computer—particularly software application developers—is a beta-tester of software that was shipped before all its defects were removed. In traditional closed-source/binary-only software deployment models, these

[1] See http://www.opensource.org for a discussion of open-source licensing terms.

premature release cycles yields frustrated users, who have little recourse when problems arise with software they purchased from suppliers. Thus, they simply find workarounds for problems they encounter and often have little incentive to contribute to improving the closed-source products.

In contrast, open-source deployment models help to leverage expertise in the user community, thereby allowing users and developers to participate together to improve software quality. A key strength of open-source is its scalability to large user communities, where application developers and end-users can assist with much of the QA, documentation, mentoring, and technical support. Throughout this paper we discuss how leveraging user communities effectively is essential to the long-term success of open-source processes and projects.

- *Scalable division of labor*. Open-source projects work by exploiting a loophole in Brooks Law[2] that states "adding people to a late project makes it later." The logic underlying this law is that as a general rule, software development productivity does *not* scale up as the number of developers increases. The culprit, of course, is the rapid increase in human communication and coordination costs as project size grows. Thus, a team of ~10 good developers can often produce much higher quality software with less effort and expense than a team of ~1,000 developers.

  In contrast, software debugging and QA productivity *does* scale up as the number of developers helping to debug the software increases. The main reason for this is that all other things being equal, having more people test the code will identify the ``error-legs" much more quickly than having just a few testers. Thus, a team of 1,000 testers should find many more bugs than a team of 10 testers. QA activities also scale better since they do not require as much inter-personal communication as software development activities (particularly design activities) often do.

  To leverage the loophole in Brooks law, therefore, most successful open-source projects have a "core" and "periphery" organizational structure. In this division of labor, a relatively small number of core developers (who may well be distributed throughout the world) are responsible for ensuring the architectural integrity of the project, e.g., they typically "vet" user contributions and bug fixes, add many new features and capabilities, and track day-to-day progress on project goals and tasks. In contrast, the periphery consists of the thousands of members of the user community who help with testing and debugging of the software released periodically by the core team. Naturally, these divisions are informal and individuals may fulfill different roles at different times during the life-cycle of an open-source project.

- *Short feedback loops*. Another reason for the success of well-organized open-source development efforts, such as Linux or ACE+TAO is the short feedback loops between the core and the periphery. Often, it's only a matter of minutes or hours from the point at which a bug is reported from the periphery to the point at which an official patch is supplied from the core to fix it. Moreover, the use of powerful Internet-enabled configuration management tools, such as the GNU Concurrent Versioning System (CVS), allows open-source users in the periphery to quickly synchronize with updates supplied by the core.

  These quick response cycles encourage an open-source user community to help with the QA process since they are "rewarded" by rapid fixes after bugs are identified. Moreover, because the source code is open for inspection, when users at the periphery do encounter bugs, they can often either help fix them directly or can provide concise test cases that allow the core developers to isolate the problem quickly. Thus, the effort of the user community extends the overall debugging effort and improves software quality rapidly.

- *Inverted stratification*. In many organizations, testers are perceived to have less status than software developers. The open-source development model inverts this stratification in many cases so that the "testers"' in the periphery are often excellent software application developers who use their considerable debugging skills when they encounter problems with the open-source base. Moreover, the open-source model makes it possible to leverage the talents of these gifted developers, who would never be satisfied to play the role of tester in a traditional software organization.

- *Greater opportunity for analysis and validation*. Open-source development techniques can also help improve software quality by enabling the use of powerful analysis and validation techniques, such as whitebox testing and model checking. Although this benefit of open-source is not widely employed today, the next-generation of testing tools and model checkers will be more effective since they can instrument and analyze large-scale systems where many components and layers (e.g., network drivers, OS, and middleware) are open-source.

In general, it is our position that closed-source software development and QA processes cannot achieve the benefits outlined above as rapidly or as cheaply as open-source processes.

Where Open-Source Succeeds and Fails

Open-source projects have clearly succeeded in many domains, particularly in infrastructure software domains, such as operating systems, compilers and language processing tools, editors, and middleware. Our experience working on open-source projects over the past 10-15 years, however, has shown that open-source works better in some contexts than in

[2] Fred Brooks, "The Mythical Man-Month," Addison-Wesley, 1975.

others. Below, we outline some of the enablers and inhibitors of open-source processes.

It is our position that the key enablers of open-source success include:

- *Communities with unmet software needs.* Certain technology communities, such as scientists or even the defense industry, have software needs that are unmet by (or are economically unappealing to) "mainstream" information technology (IT) software markets. These communities often lack sufficient market presence to be well supported by mass-market IT vendors, such as Microsoft or Sun. For example, Linux-based Beowulf clusters were developed in the scientific computing community in part because their high-end computing applications run on hardware and infrastructure software configurations that are not widely supported via mass-market IT vendors.
- *Technically sophisticated user community*. Members of certain user communities have considerable software development skills and knowledge of common development tools, such as compilers, debuggers, configuration managers, online bug tracking systems, memory leak/validation tools, and performance profilers. Likewise, they are facile with rapid and open communication mechanisms, such as web/ftp-sites, mailing lists, and are familiar with advanced Internet collaboration mechanisms, such as Netmeeting or Instant Messaging.
- *General-purpose, commoditized, horizontal infrastructure software*, where requirements and APIs are well known. For example, the requirements and APIs for C/C++ compilers/linkers, UNIX/POSIX-based operating systems, HTTP Web servers, and CORBA object request broker (ORB) middleware are all well understood by conventionally trained software developers. Thus, there is little need to expend time and effort on costly upstream software development activities, such as requirements analysis or interface specifications. Moreover, there is an abundance of highly talented graduate students and programming aficionados who enjoy working on these types of general-purpose software infrastructure technologies.

It is also our position, however, that open-source may not be as effective or desirable in other contexts, such as

- *Highly vertical domains*. Certain domains, such as distributed electronic medical imaging or enterprise resource planning, require highly specialized technical skills and are not standardized. As a result, it is hard to establish an open-source developer community since graduate students and programming aficionados rarely have sufficient domain-knowledge or interest to work on these topics.
- *Highly competitive domains*. In some domains, such as online trading systems, the financial incentives are so high that developers in these domains are not motivated to make their work available for open-source use by competitors.
- *Highly secure domains.* In other domains, such theater ballistic missile tracking systems, security issues are paramount and there are national security/export laws that restrict the dissemination of software.
- *High-confidence domains*. In safety-critical domains, such as nuclear power plan process control or civil aviation flight-control, there are strict certification procedures that must be followed to validate that software is sufficiently robust for mission-critical operations. The development and QA processes for most open-source projects today lack sufficient rigor to pass these certification tests.

While we believe that it may ultimately be possible to apply technology solutions to address concerns with using open-source in high-confidence domains, the business and security issues may preclude open-source from being adopted in the other three types of domains outlined above.

Technical Challenges for Open-Source Developers

Despite the many strengths of open-source processes, developers of open-source software face a number of challenges, including:

- *Long-term maintenance and evolution costs*. Due to their minimal/non-existent licensing fees, open-source projects often run on a "shoe-string" QA budget, which makes it impractical to support large numbers of versions and variants simultaneously.
- *Quality assurance*. It is hard to ensure consistent quality of open-source software as a result of the following characteristics of open-source projects:
  - *Frequent beta releases*. As mentioned earlier, a cornerstone of open-source is short feedback loops, which typically result in frequent "beta" releases, e.g., several times a month. Although this schedule satisfies end-users who have found bugs in preceding betas, it can frustrate other end-users who want more stable software releases.
  - *Heterogeneous platforms*. Another cornerstone of open-source software is its platform-independent. This feature can yield the Sisyphean task, however, of keeping an open-source source software base operational despite continuous changes to the underlying OS and compiler platforms.
  - *Support for many compile-time and run-time configurations*. The availability of the software in open-source project encourages core developers to increase the number of options for configuring and subsetting the software at compile-time and run-time. Although this enhances the software's applicability for a broad range of use-cases, it also greatly magnifies QA costs due to

the combinatorial number of code paths that must be regression tested.

In general, the QA challenges of open-source software are to simultaneously limit regression errors (i.e., breaking features that worked in previously releases), sustain end-user confidence and good will, and minimize development & QA costs.

- *Ensuring coherency of system-wide properties*. The decentralized nature of many open-source projects makes it hard to enforce a common "look and feel," standardize common APIs, and ensure consistent semantics when disparate components are integrated together. Since open-source projects often grow "organically" based on contributions from a range of developers and users, considerable effort must be expended to ensure architectural integrity and commonality across contributions from disparate members of the community.

Overview of the Skoll Project

Skoll Goals

We have begun a long-term, multi-site collaborative research project, called Skoll, that is leveraging open-source assets (such as technologically sophisticated worldwide user communities, open access to source, and ubiquitous web access) to address the key challenges of open-source software development outlined above. The goals of the Skoll project are to:

- *Incrementally improve the quality and performance of open-source systems*. Keeping with the spirit of open-source efforts, our aim is to opportunistically improve the software base over time, though not necessarily perfectly at each step.
- *Minimize human effort by applying automation*. We are automating the role of human "build czars," who today monitor the stability of open-source software repositories manually to ensure problems are fixed rapidly.
- *Minimize unnecessary end-user overhead*. We are using automated web tools to minimize end-user effort and resource commitments, as well as ensure the security of end-user computing sites.

In Skoll, we are also leveraging the openness of the source code as an "enabling technology" to

- *Detect and resolve QA problems quickly*. To close the loop from users back to developers rapidly, we are exploiting the inherent distributed nature of open-source sites/users, each one performing a portion of the overall testing to offload the number of versions that must be maintained by the core developers, while simultaneously enhancing user confidence in new beta versions of open-source software.
- *Automatically analyze and enhance key system QoS characteristics on multiple platforms*. Each site/user conducts different instrumentations and performance benchmarks automatically to collect (1) static and dynamic memory footprint metrics and (2) throughput, latency, and jitter performance metrics.
- *Understand and enhance key system usability characteristics*, such as usage errors.

The approach we are taking to achieve these goals is based on the notion of *continuous testing*, which is operationalized in Skoll as follows:

- Regression testing and performance profiling is widely distributed and conducted in parallel on machines provided by the open-source user community during their off-peak hours;
- The resources of the user community are coordinated carefully, i.e., we follow the sun around the world[3] and adapting the testing/profiling processes based on results of earlier testing and profiling.

We are pursuing these goals in the context of the ACE[4] and TAO[5] open-source projects centered at the University of California, Irvine and Washington University, St. Louis. ACE is a widely-used OO framework containing a rich set of components that implement patterns for high-performance and real-time communication systems. TAO is an implementation of CORBA that uses the framework components in ACE to meet the demanding quality of service requirements in distributed, real-time, and embedded systems. These two projects are ideal study candidates because they share important characteristics with other open-source projects, such as

- *Large user community*, e.g., more than 20,000 users who work for thousands of companies in hundreds of countries worldwide.
- *Large code base*, i.e., over a million source lines of complex C++ middleware systems code maintained and enhanced by a core team of ~30 developers worldwide.
- *Continuous evolution*. i.e., a dynamically changing and growing code base that has an average of 200+ CVS repository commits per week.
- *Highly heterogeneous*. ACE+TAO run on dozens of OS and compiler platforms. These platforms change frequently over time, e.g., to support new versions of the C++ standard, different versions of POSIX/UNIX, and updates to over a dozen other non-UNIX OS platforms, including

[3] This is the origin of the term "Skoll," which is a Scandinavian myth that explains the sunrise and sunset cycles.
[4] ACE is available at http://www.cs.wustl.edu/~schmidt/ACE.html
[5] TAO is available at http://www.cs.wustl.edu/~schmidt/TAO.html

all variants of Microsoft Win32, as well as many real-time operating systems.

- *Highly configurable*, e.g., numerous interdependent options supporting wide variety of "use-case" families and standards, such as standard CORBA, Minimum CORBA, Real-time CORBA, and many different CORBA services.

The team of ~30 core developers of ACE and TAO automatically regression test these systems continuously on the 40 or so workstations and servers at their respective sites in Irvine, California and St. Louis, Missouri. They cannot test all possible platform and OS combinations, however, because there simply are not enough people, OS/compiler, platforms, CPU cycles, or disk space to run all the hundreds of tests in a timely manner. Moreover, ACE+TAO are designed for ease of subsetting, i.e., there are several hundred orthogonal features/options that can be enabled for a particular use-case. Thus, there are a combinatorial number of possible configurations that could be tested at any given point, which is far beyond the resources of a core open-source development team to manage.

We believe that ACE+TAO are representative of many other large-scale open-source projects, such as Linux, the GNU compiler and language processing tools, Perl, and Apache. Therefore, developing a scalable methodology that can improve the quality and performance of ACE+TAO via continuous testing and profiling should help these other open-source projects, as well.

Skoll Experiment Structure

We are designing experiments using ACE and TAO as our subject programs. Our intent is to enlist ACE+TAO users as study participants to perform the following activities:

1. Use a web registration form through which study participants can describe their platforms, i.e., their operating system and their compiler/linker characteristics.
2. Use a mailing list to communicate with study participants (ACE+TAO users already communicate readily with the core developers via several heavily trafficed mailing lists).
3. Distribute a portable Perl script to automatically download ACE+TAO from the main CVS repository where it resides at Washington University.
4. At periodic intervals (designated by the study participants) send every participant a configuration file that is customized for their platform. To provide coverage of the entire space of possible configuration options, this configuration file will be generated randomly—based on rules that ensure its semantic validity.
5. Participants will compile ACE+TAO using this configuration, run all the tests, and send any problems back to the core developers or link it into a “virtual scoreboard” that can be examined later via a Web browser. Each core developer responsible for different components of ACE+TAO will have their own view into the problem results or virtual scoreboard so that they know what to fix.
6. Steps 1 through 5 will be repeated continuously, running two types of tests:
    a. Tests to evaluate the syntactic and semantic correctness of ACE+TAO with respect to the generated configuration. Syntactic checks will be performed at compile-time. More advanced semantic checks will be performed by running regression tests. Two different approaches are reserved for the run-time problems
        i. Maximize diversity to find the greatest range of problems and
        ii. Focus the available resources to pinpoint the problem when problems are found.
    b. Tests that collect metrics on performance, footprint, etc., to pinpoint when and why ACE+TAO’s performance decreases on particular platforms.

Based on the results of the initial study outlined above, we plan to conduct more advanced studies that will use results and feedback from temporally earlier experiments to focus subsequent tests (and subsequent versions) that are distributed around the world. For example, results of tests run in India can be used to guide the configurations tested in Europe six hours later. We also plan to automate error detection (e.g., via CVS rollback capabilities) and provide tools that will recommend a course of action to human build czars if errors cannot be detected automatically.

Concluding Remarks

Historically, software quality has lagged behind hardware quality. Over the past decade, however, the maturation of R&D efforts on patterns, frameworks, components, and middleware has enabled open-source technologies like ACE and TAO to be used in an increasing number of mission-critical infrastructure applications, such as avionics mission computing, hot rolling mills, backbone routers, and high-speed network switches, and pervasive computing applications, such as wireless and wireline phones, PDAs, and other consumer electronic devices. As the open-source community continues to improve the quality and ubiquity of its technologies, we believe its presence will increase in many, though perhaps not all, application domains. The Skoll project outline in this position paper is leveraging key aspects of open-source, such as technologically sophisticated worldwide user communities, open access to source, and ubiquitous web access. This project offers an opportunity to apply advanced validation and performance improvement techniques throughout many levels of complex software systems.