13 Adopting Open Source Software Engineering (OSSE) Practices by Adopting OSSE Tools

Jason Robbins

The open source movement created a set of software engineering tools with features that fit the characteristics of open source development processes. To a large extent, the open source culture and methodology are conveyed to new developers via the toolset itself and the demonstrated usage of these tools on existing projects. The rapid and wide adoption of open source tools stands in stark contrast to the difficulties encountered in adopting traditional Computer-Aided Software Engineering (CASE) tools. This chapter explores the characteristics that make these tools adoptable and discusses how adopting them may influence software development processes.

One ongoing challenge facing the software engineering profession is the need for average practitioners to adopt powerful software engineering tools and methods. Starting with the emergence of software engineering as a field of research, increasingly advanced tools have been developed to address the difficulties of software development. Often these tools addressed accidental difficulties of development, but some have been aimed at essential difficulties such as management of complexity, communication, visibility, and changeability (Brooks 1987). Later, in the 1990's, the emphasis shifted from individual tools toward the development process in which the tools were used. The software process movement produced good results for several leading organizations, but it did not have much impact on average practitioners.

Why has adoption of CASE tools been limited? Often the reason has been that they did not fit the day-to-day needs of the developers who were expected to use them: they were difficult to use, expensive, and special purpose. The fact that they were expensive and licensed on a per-seat basis caused many organizations to only buy a few seats, thus preventing other members of the development team from accessing the tools and artifacts only available through these tools. One study of CASE tool adoption found

that adoption correlates negatively with end user choice, and concludes that successful introduction of CASE tools must be a top-down decision from upper management (Iivari 1996). The result of this approach has repeatedly been "shelfware": software tools that are purchased but not used.

Why have advanced methodologies not been widely adopted? Software process improvement efforts built around capability maturity model (CMM) or ISO-9000 requirements have required resources normally only found in larger organizations: a software process improvement group, time for training, outside consultants, and the willingness to add overhead to the development process in exchange for risk management. Top-down process improvement initiatives have often resulted in a different kind of shelfware, where thick binders describing the organization's software development method (SDM) go unused. Developers who attempt to follow the SDM may find that it does not match the process assumptions embedded in current tools. Smaller organizations and projects on shorter development cycles have often opted to continue with their current processes or adopt a few practices of lightweight methods such as extreme programming in a bottom-up manner (Beck 2000).

In contrast, open source projects are rapidly adopting common expectations for software engineering tool support, and those expectations are increasing. Just four years ago, the normal set of tools for an open source project consisted of a mailing list, a *bugs* text file, an *install* text file, and a CVS server. Now, open source projects are commonly using tools for issue tracking, code generation, automated testing, documentation generation, and packaging. Some open source projects have also adopted object-oriented design and static analysis tools. The feature sets of these tools are aimed at some key practices of the open source methodology, and in adopting the tools, software developers are predisposed to also adopt those open source practices.

Exploring and encouraging development and adoption of open source software engineering tools has been the goal of the http://tigris.org Web site for the past three years. The site hosts open source projects that are developing software engineering tools and content of professional interest to practicing software engineers. Tigris.org also hosts student projects on any topic, and a reading group for software engineering research papers. The name "Tigris" can be interpreted as a reference to the Fertile Crescent between the Tigris and Euphrates rivers. The reference is based on the hypothesis that an agrarian civilization would and did arise first in the location best suited for it. In other words, the environment helps define

the society, and more specifically, the tools help define the method. This is similar to McLuhan's proposition that "the medium is the message" (McLuhan 1994).

Some Practices of OSS and OSSE

The open source movement is broad and diverse. Though it is difficult to make generalizations, there are several common practices that can be found in many open source software projects. These practices leave their mark on the software produced. In particular, the most widely adopted open source software engineering tools are the result of these practices, and they embody support for the practices, which further reinforces the practices.

Tools and Community

Provide Universal, Immediate access to All Project Artifacts The heart of the open source method is the accessibility of the program source code to all project participants. Beyond the source code itself, open source projects tend to allow direct access to all software development artifacts such as requirements, design, open issues, rationale, development team responsibilities, and schedules. Tools to effectively access this information form the centerpiece of the open source development infrastructure: projects routinely make all artifacts available in real time to all participants worldwide over the Internet. Both clients and server components of these tools are available on a wide range of platforms at zero cost. This means that all participants can base their work on up-to-date information. The availability of development information is also part of how open source projects attract participants and encourage them to contribute.

In contrast, traditional software engineering efforts have certainly made progress in this area, but it is still common to find projects that rely on printed binders of requirements that rapidly become outdated, use LAN-based collaboration tools that do not scale well to multisite projects, purchase tool licenses for only a subset of the overall product team, and build silos of intellectual property that limit access by other members of the same organization who could contribute. While e-mail and other electronic communications are widely used in closed source projects, the information in these systems is incomplete because some communication happens face-to-face or via documents that are never placed in a shared repository.

Staff Projects with Motivated Volunteers Open source projects typically have no dedicated staff. Instead, work is done by self-selected developers who volunteer their contributions. Self-selection is most likely to occur when the developers are already familiar with the application domain and development technologies. Developers allocate their own time to tasks that they select. This means that every feature is validated by at least one person who strongly desired it. Motivation for open source development comes in many forms, including one's own need for particular software, the joy of construction and expression, altruism, the need for external validation of one's own ideas and abilities, the ideology of free software as a form of freedom, and even social and financial rewards. The other side of joy as a motivation is that unlikable jobs tend to go undone, unless they are automated. While some high-profile open source projects have ample potential contributors, a much larger number of average open source projects rely on the part-time efforts of only a few core members.

In contrast, traditional software engineering projects are staffed and funded. Often organizations emphasize continuity and stability as ways to keep costs down over the life of a product line. Achieving staff continuity in a changing business and technology environment demands that training be part of the critical path for many projects. Traditional software engineers are motivated by many of the same factors found in open source, as well as professionalism and direct financial incentives. Resources are always limited, even in well-funded commercial projects, and it is up to management to determine how those resources are allocated.

Work in Communities that Accumulate Software Assets and Standardize Practices Collaborative development environments (CDEs) such as SourceForge[1] and SourceCast[2] now host large development communities that would have previously been fragmented across isolated projects hosted on custom infrastructure. This is one of the most important shifts in open source development. It was partly inspired by the Mozilla.org toolset, which itself descended from a more traditional software engineering environment. These large development communities reduce the effort needed to start a new project by providing a complete, standard toolset. They warehouse reusable components, provide access to the developers that support them, and make existing projects in the communities accessible as demonstrations of how to use those tools and components. Preference for standards and conventions is particularly strong in the selection of tools in open source projects. Increasingly, it is the development community as a whole that has made decisions about the set of tools in the

CDE, and individual projects accept the expectation of using what is provided. In particular, a great increase in the reliance on issue-tracking tools by open source projects has resulted from the availability and demonstrated usage of issue trackers in CDEs.

Many larger commercial software development organizations do have organization-wide standards and site licenses for fundamental tools such as version control. However, it is still common for projects to acquire licenses for specific tools using a project-specific budget with little standardization across projects. Software process improvement (SPI) teams have attempted to standardize practices through training, mandates, and audits. However, they have rarely been able to leverage the visibility of best practices across projects. Peer visibility is an important key to making a methodology become ingrained in a development culture. Likewise, providing a repository of reusable components is not, in itself, enough to drive reuse: developers look for evidence that others are successfully reusing a given component.

Open Systems Design

Follow Standards to Validate the Project, Scope Decision Making, and Enable Reuse A preference for following standards is deeply ingrained in the open source culture. The need for pre-1.0 validation of the project and the lack of formal requirements generation in open source projects tends to encourage reliance on externally defined standards and conventions. Deviation from standards is discouraged because of the difficulty of specifying an alternative with the same level of formality and agreement among contributors. Standards also define interfaces that give choice to users and support diversity of usage.

Standards and open systems are also emphasized in traditional development projects. The current move to web services is one important example of that. However, the marketplace often demands that new products differentiate themselves from existing offerings by going beyond current standards. At the same time, pressure to maximize returns may justify a decision to implement only part of a standard and then move on to other revenue-generating functionality.

Practice Reuse and Reusability to Manage Project Scope Open source projects generally start with very limited resources, often only one or two part-time developers. Projects that start with significant reuse tend to be more successful, because they can demonstrate results sooner, they focus

discussions on the project's value-added, and they resonate with the cultural preference for reuse. Even if a project had implemented its own code for a given function, peer review often favors the elimination of that code, if a reusable component can replace it. Reusable components can come from projects that explicitly seek to create components for use by developers, or they can spin out of other projects that seek to produce end user products. In fact, spinning out a reusable component is encouraged, because it fits the cultural preference for reuse, and often gives a mid-level developer the social reward of becoming a project leader.

The return on building reusable components can be hard to estimate in advance. So the justification for reusable components in traditional software development may be unclear, even in organizations with reuse initiatives. In contrast, the motivations for open source participation apply to the development of components as much or more than they do to the development of end user products. Traditional development teams are responsible for maximizing returns on their current project; the cost of providing ongoing support for reusable components can be at odds with that goal. In contrast, open source components can achieve a broad population of users that can support one another.

Support Diversity of Usage and Encourage Plurality of Authorship Open source products are often cross-platform and internationalized from the start. They usually offer a wide range of configuration options that address diverse use cases. Any contributor is welcome to submit a new feature to "scratch an itch" (Raymond 2001). Such willingness to add functionality can lead to feature creep and a loss of conceptual integrity. This sort of occurrence can make it harder to meet predefined deadlines, but it broadens the appeal of the product, because more potential users get their own win conditions satisfied. Since users are responsible for supporting each other, the increase in the user population can provide the increased effort needed to support the new features. Peer review, standards, and limited resources can help limit undirected feature creep.

While traditional development tools may have great depth of functionality, they tend to have fewer options and more platform restrictions than their open source counterparts, making it harder for large organizations to select a single tool for all development efforts across the enterprise. Commercial development projects manage a set of product features in an effort to maximize returns while keeping support costs under control. Likewise, management assigns specific tasks to specific developers and holds them accountable for those tasks, usually to the exclusion of serendipitous con-

tributions. Even if an outside contributor submitted a new piece of functionality, the cost of providing technical support for that functionality may still prevent its integration.

Planning and Execution

Release Early, Release Often Open source projects are not subject to the economic concerns or contractual agreements that turn releases into major events in traditional development. For example, there are usually no CDs to burn and no paid advertising campaigns. That reduced overhead allows them to release as early and often as the developers can manage. A hierarchy of release types is used to set user expectations: "stable" releases may happen at about the same rate as releases in traditional development, but "nightly" releases are commonly made available, and public "development" releases may happen very soon after the project kickoff and every few weeks thereafter. In fact, open source projects need to release pre-1.0 versions in order to attract the volunteer staff needed to reach 1.0. But, a rush toward the first release often means that traditional upstream activities such as requirements writing must be done later, usually incrementally. Reacting to the feedback provided on early releases is key to requirement-gathering and risk-management practices in open source.

In contrast, a traditional waterfall development model invests heavily in upstream activities at the start of the project in an attempt to tightly coordinate work and minimize the number of releases. Many organizations have adopted iterative development methodologies, for example, extreme programming (Beck 2000) or "synch and stabilize" (Cusumano and Selby 1995). However, they still must achieve enough functionality to have a marketable 1.0 release. And concerns about exposing competitive information and the overhead of integration, training, marketing, and support create a tendency toward fewer, more significant releases.

Place Peer Review in the Critical Path Feedback from users and developers is one of the practices most central to the open source method. In many open source projects, only a core group of developers can commit changes to the version control system; other contributors must submit a patch that can be applied only after review and discussion by the core developers. Also, it is common for open source projects to use automated email notifications to prompt broad peer review of each CVS commit. Peer review has also been shown to be one of the most effective ways to eliminate defects in code, regardless of methodology (Wiegers 2002). The claim that

"given enough eyeballs, all bugs are shallow" (Raymond 2001, 41) underscores the value of peer review, and it has proven effective on some high profile open source projects. However, unaided peer review by a few average developers, who are for the most part the same developers who wrote the code in the first place, is not a very reliable or efficient practice for achieving high quality.

Although the value of peer reviews is widely acknowledged in traditional software engineering, it is unlikely to be placed in the critical path unless the project is developing a safety-critical system. Traditional peer reviews require time for individual study of the code followed by a face-to-face review meeting. These activities must be planned and scheduled, in contrast to the continuous and serendipitous nature of open source peer review.

Some Common OSSE Tools

This section reviews several open source software engineering tools with respect to the practices defined previously. Editors, compilers, and debuggers have not been included; instead, the focus is on tools that have more impact on collaborative development. Most of these tools are already widely used, while a few are not yet widely used but are set to rapidly expand in usage.

Version Control

CVS, WinCVS, MacCVS, TortoiseCVS, CVSWeb, and ViewCVS The Concurrent Versions System (CVS) is the most widely used version control system in open source projects. Its features include a central server that always contains the latest versions and makes them accessible to users over the Internet; support for disconnected use (i.e., users can do some work while not connected to the Internet); conflict resolution via merging rather than locking to reduce the need for centralized coordination among developers; simple commands for checking in and out that lower barriers to casual usage; cross-platform clients and servers; and, a vast array of options for power users. It is common for CVS to be configured to send e-mail notifications of commits to project members to prompt peer review. WinCVS, MacCVS, and TortoiseCVS are just three of many free clients that give users a choice of platform and user interface style. CVS clients are also built into many IDEs and design tools. CVSWeb and ViewCVS are Web-based tools for browsing a CVS repository.

Adoption of CVS among open source projects is near total, and the concepts embodied in CVS have clearly influenced the open source methodology. CVS can easily provide universal access to users of many platforms and many native languages at locations around the globe. The practice of volunteer staffing takes advantage of CVS's straightforward interface for basic functions, support for anonymous and read-only access, patch creation for later submission, and avoidance of file locking. CVS has been demonstrated to scale up to large communities, despite some shortcomings in that regard. The protocol used between client and server is not a standard; however, CVS clients have followed the user interface standards of each platform. In fact, the command-line syntax of CVS follows conventions established by the earlier RCS system. A separation of policy from capability allows a range of branching and release strategies that fit the needs of diverse projects. Frequent releases and hierarchies of release quality expectations are facilitated by CVS's ability to maintain multiple branches of development. Peer review is enabled by easy access to the repository, and is encouraged by email notifications of changes.

Subversion, RapidSVN, TortoiseSVN, and ViewCVS Subversion is the leading successor to CVS. Its features include essentially all of CVS's features, with several significant enhancements: it has a cleaner, more reliable, and more scalable implementation; it is based on the existing WebDAV standard; it replaces CVS's concepts of branches and tags with simple naming conventions; and, it has stronger support for disconnected use. RapidSVN and TortoiseSVN are two of several available Subversion clients. ViewCVS can browse Subversion repositories as well as CVS repositories. Also, Subversion repositories can be browsed with any standard Web browser and many other applications, due to the use of the standard WebDAV protocol.

It will take time for Subversion to be widely adopted by open source projects, but interest has already been very high and many early uses have been documented. Subversion improves on CVS's support for universal access by following standards that increase scalability and ease integration. Diverse users already have a choice of several Subversion clients; however, there are fewer than those of CVS. Subversion's simplification of branching lowers the learning curve for potential volunteers and supports a diversity of usage. Support for frequent releases and peer review in Subversion is similar to that of CVS.

Issue Tracking and Technical Support

Bugzilla Bugzilla was developed to fit the needs of the Mozilla open source project. Its features include: an "unconfirmed" defect report state needed for casual reporters who are more likely to enter invalid issues; a "whine" feature to remind developers of issues assigned to them; and a Web-based interface that makes the tool cross-platform, universally accessible, and that lowers barriers to casual use.

Bugzilla has been widely adopted and deeply integrated into the open source community over the past few years. The Bugzilla database on Mozilla.org has grown past 200,000 issues, and a dozen other large open source projects each host tens of thousands of issues. The whine feature helps address the lack of traditional management incentives when projects are staffed by volunteers. Bugzilla has been demonstrated to scale up to large communities, and the organized history of issues contained in a community's issue database serve to demonstrate the methodology practiced by that community. When developers evaluate the reusability of a component, they often check some of the issues in the project issue tracker and look for signs of activity. Conversely, when developers feel that they have no recourse when defects are found in a reusable component, they are likely to cease reusing it. There is a remarkable diversity of usage demonstrated in the issues of large projects: developers track defects, users request support, coding tasks are assigned to resources, patches are submitted for review, and some enhancements are debated at length. Frequent releases and peer review of project status are enabled by Bugzilla' clear reporting of the number of pending issues for an upcoming release.

Scarab The Scarab project seeks to establish a new foundation for issue-tracking systems that can gracefully evolve to fit many needs over time. Scarab covers the same basic features as does Bugzilla, but adds support for issue de-duplication on entry to defend against duplicates entered by casual participants; an XML issue exchange format; internationalization; and highly customizable issue types, attributes, and reports.

Interest and participation in the Scarab project has been strong, and the tool is rapidly becoming ready for broader adoption. Scarab's support for internationalization and XML match the open source practices of universal access and preference for standards and interoperability. Scarab's biggest win comes from its customizability, which allows the definition of new issue types to address diverse user needs.

Technical Discussions and Rationale

Mailing Lists Mailing lists provide a key advantage over direct e-mail, in that they typically capture messages in Web-accessible archives that serve as a repository for design and implementation rationale, as well as end user support information. Some of the most common usages of mailing lists include: question and answer sessions among both end users and developers, proposals for changes and enhancements, announcements of new releases, and voting on key decisions. Voting is often done using the convention that a message starting with the text "+1" is a vote in favor of a proposal, a message with "+0" or "–0" is an abstention with a comment, and a message with "–1" is a veto, which must include a thoughtful justification. While English is the most commonly used natural language for open source development, mailing lists in other languages are also used.

Open source developers adopted mailing lists early, and now they are used on very nearly every project. Since mailing lists use e-mail, they are standards-based, cross-platform, and accessible to casual users. Also, since the e-mail messages are free-format text, this single tool can serve a very diverse range of use cases. It is interesting to note that the preference is for plain-text messages: HTML-formatted e-mail messages and integration of e-mail with other collaboration features have not been widely adopted. Project mailing list archives do help set the tone of development communities, but the flexibility of mailing lists allows so many uses that new potential developers might not recognize many of the patterns. Peer review usually happens via mailing lists, because CVS's change notifications use e-mail, and because e-mail is the normal medium for discussions that do not relate to specific issues in the issue database.

Project Web Sites Open source predates the Web, so early open source projects relied mainly on mailing lists, file transfer protocol (FTP), and later, CVS. Open source projects started building and using Web sites soon after the introduction of the Web. In fact, several key open source projects such as Apache are responsible for significant portions of today's Web infrastructure. A typical open source Web site includes a description of the project, a users' guide, developer documentation, the names of the founding members and core developers, the license being used, and guidelines for participation. Open source Web sites also host the collaborative development tools used on the project. Users can find related projects by following links from one individual project to another, but

increasingly, projects are hosted at larger community sites that categorize related projects.

Project Web sites have been universally adopted by recent open source projects. Web sites provide universal access to even the most casual users. Web page design can adjust to suit a wide range of diverse uses and preferences. The temptation to build an unusual Web site for a particular project is sometimes in conflict with the goals of the larger community site. Community-wide style sheets and page design guidelines reduce this conflict, as do tools like Maven and SourceCast that define elements of a standard appearance for each project's Web content. Internet search engines enable users to find open source products or reusable components. The Web supports the practice of issuing frequent releases simply because the project's home page defines a stable location where users can return to look for updates. Also, Internet downloads support frequent reuse by eliminating the need for printed manuals, CDs, packaging, and shipping.

HOWTOs, FAQs, and FAQ-O-Matic HOWTO documents are goal-oriented articles that guide users through the steps needed to accomplish a specific task. Lists of frequently asked questions (FAQs) help to mitigate two of the main problems of mailing lists: the difficulty of summarizing the discussion that has gone before, and the wasted effort of periodically revisiting the same topics as new participants join the project. FAQ-O-Matic and similar tools aim to reduce the unlikable effort of maintaining the FAQ.

FAQs and HOWTOs are widely used, while FAQ-O-Matic is not nearly so widely used. This may be the case because simple HTML documents serve the purpose and allow more flexibility. FAQs and HOWTOs are universally accessible over the Internet, and tend to be understandable by casual users because of their simple, goal-oriented format. Developer FAQs and HOWTOs can help potential volunteers come up to speed on the procedures needed to make specific enhancements. When FAQ-O-Matic is used, it helps reduce the tedious task of maintaining a FAQ, and makes it easier for users to suggest that new items be added. Many HOWTO documents conform to a standard SGML document type and are transformed into viewable formats by using DocBook or other tools.

Wiki, TWiki, and SubWiki A *wiki* is a collaborative page-editing tool in which users may add or edit pages directly through their web browser. Wikis use a simple and secure markup language instead of HTML. For example, a word written like "NameOfPage" would automatically link to another page in the wiki system. Wikis can be more secure than systems

that allow entry of HTML, because there is no way for users to enter potentially dangerous JavaScript or to enter invalid HTML markup that could prevent the overall page from rendering in a browser. TWiki is a popular wiki-engine with support for page histories and access controls. SubWiki is a new wiki-engine that stores page content in Subversion.

Wiki-engines are used in many open source projects, but by no means in a large fraction of all open source projects. Wiki content is universally accessible over the Web. Furthermore, volunteers are able to make changes to the content without the need for any client-side software, and they are sometimes even free to do so without any explicit permission. Wiki sites do have a sense of community, but Wiki content tends to serve as an example of how to loosely organize pages of documentation, rather than a demonstration of any particular development practice.

Build Systems

Make, Automake, and Autoconf The Unix "make" command is a standard tool to automate the compilation of source code trees. It is one example of using automation to reduce barriers to casual contributors. And there are several conventions that make it easier for casual contributors to deal with different projects. Automake and Autoconf support portability by automatically generating makefiles for a particular Unix environment.

These tools are widely used; in fact, if it were not for Ant (see the following section) the use of make would still be universal. While makefiles are not particularly easy to write or maintain, they are easy for users and volunteer developers to quickly learn to run. Makefiles are based on a loose standard that dates back to the early days of Unix. Developers who intend to reuse a component can safely assume that it has a makefile that includes conventional make targets like "make clean" and "make install." Makefiles are essentially programs themselves, so they can be made arbitrarily complex to support various diverse use cases. For example, in addition to compiling code, makefiles can be used to run regression tests. Running regression tests frequently is one key to the practice of frequent releases.

Ant Ant is a Java replacement for make that uses XML build files instead of makefiles. Each build file describes the steps to be carried out to build each target. Each step invokes a predefined task. Ant tasks each perform a larger amount of work than would a single command in a makefile. This process can reduce the tedium of managing complex makefiles, increase

consistency across projects, and ease peer review. In fact, many projects seem to use build files that borrow heavily from other projects or examples in the Ant documentation. Many popular IDEs now include support for Ant.

Ant is being adopted rapidly by both open source and traditional software development projects that use Java. Ant is accessible to developers on all platforms, regardless of whether they prefer the command-line or an IDE. Ant build files tend to be more standard, simpler, and thus more accessible to potential volunteers. Since Ant build files are written in XML, developers are already familiar with the syntax, and tools to edit or otherwise process those files can reuse standard XML libraries. As Ant adoption increases, developers evaluating a reusable Java component can increasingly assume that Ant will be used and that conventional targets will be included. As with make, Ant can be used for regression testing to support the practice of delivering frequent releases.

Tinderbox, Gump, CruiseControl, XenoFarm, and Maven Nightly build tools automatically compile a project's source code and produce a report of any errors. In addition to finding compilation errors, these tools can build any make or Ant target to accomplish other tasks, such as regression tests, documentation generation, or static source code analysis. These tools can quickly catch errors that might not have been noticed by individual developers working on their own changes. Some nightly build tools automatically identify and notify the developer or developers who are responsible for breaking the build, so that corrections can be made quickly. Tinderbox and XenoFarm can also be used as "build farms" that test the building and running of the product on an array of different machines and operating systems.

Nightly build tools have been used within large organized communities such as Mozilla.org and Jakarta.apache.org, as well as by independent projects. These tools help provide universal access to certain important aspects of the project's current status: that is, does the source code compile and pass unit tests? Volunteers may be more attracted to projects with clear indications of progress than they would otherwise. And the limited efforts of volunteers need not be spent on manually regenerating API documentation or running tests. Component reuse is encouraged when developers can easily evaluate the status of development. Organized open source development communities use nightly build tools to help manage dependencies between interdependent projects. Frequent releases are facilitated

by nightly build automation that quickly detects regressions and notifies the responsible developers.

Design and Code Generation

ArgoUML and Dia ArgoUML is a pure-Java UML design tool. ArgoUML closely follows the UML standard, and associated standards for model interchange and diagram representation. In addition to being cross-platform and standards based, it emphasizes ease of use and actively helps train casual users in UML usage. ArgoUML's design critics catch design errors in much the same way that static analysis tools catch errors in source code. ArgoUML is one of very few tools to support the Object Constraint Language (OCL), which allows designers to add logical constraints that refine the meaning of their design models. Dia is a more generic drawing tool, but it has a UML mode that can also generate source code.

UML modeling tools have experienced only limited adoption among open source projects, possibly because of the emphasis on source code as the central development artifact. Tools such as ArgoUML that are themselves open source and cross-platform provide universal access to design models, because any potential volunteer is able to view and edit the model. Emphasis on standards in ArgoUML has allowed for model interchange with other tools and the development of several plug-ins that address diverse use cases. If design tools were more widely used in open source projects, UML models would provide substantial support for understanding components prior to reuse, for peer reviews at the design level, and for the sharing of design patterns and guidelines within development communities.

Torque, Castor, and Hibernate Torque is a Java tool that generates SQL and Java code to build and access a database defined by an XML specification of a data model. It is cross-platform, customizable, and standards-based. Torque's code generation is customizable because it is template-based. Also, a library of templates has been developed to address incompatibilities between SQL databases. Castor addresses the same goals, but adds support for persistence to XML files and parallels relevant Java data access standards. Hibernate emphasizes ease of use and rapid development cycles by using reflection rather than code generation.

Open source developers have adopted database code generation tools at about the same rate that traditional developers have. Projects that have

adopted them are able to produce products that are themselves portable to various databases. Code generation tools can increase the effectiveness of volunteer developers and enable more frequent releases by reducing the unlikable tasks of writing repetitive code by hand, and debugging code that is not part of the project's value-add. Code generation is itself a form of reuse in which knowledge about a particular aspect of implementation is discussed by community members and then codified in the rules and templates of the generator. Individual developers may then customize these rules and templates to fit any unusual needs. Peer review of schema specifications can be easier than reviewing database access code.

XDoclet, vDoclet, JUnitDoclet, and Doxygen These code generation tools build on the code commenting conventions used by Javadoc to generate API documentation. Doxygen works with C, C++, IDL, PHP, and C#, in addition to Java. XDoclet, vDoclet, and JUnitDoclet can be used to generate additional code rather than documentation. For example, a developer could easily generate stubs for unit tests of every public method of a class. Another use is the generation of configuration files for Web services, application servers, or persistence libraries. The advantage of using comments in code rather than an independent specification file is that the existing structure of the code is leveraged to provide a context for code generation parameters.

Like the code generators listed previously, doclet-style generators are a form of reuse that reduces the need to work on unlikable tasks, and output templates can be changed to fit the needs of diverse users. The doclet approach differs from other code generation approaches in that no new specification files are needed. Instead, the normal source code contains additional attributes used in code generation. This is a good match for the open source tendency to emphasize the source code over other artifacts.

Quality Assurance Tools

JUnit, PHPUnit, PyUnit, and NUnit JUnit supports Java unit testing. It is a simple framework that uses naming conventions to identify test classes and test methods. A test executive executes all tests and produces a report. The JUnit concepts and framework have been ported to nearly every programming language, including PHPUnit for PHP, PyUnit for Python, and NUnit for C#.

JUnit has been widely adopted in open source and traditional development. It has two key features that address the practices of the open source

methodology: test automation, which helps to reduce the unlikable task of manual testing that might not be done by volunteers; and unit test reports, which provide universally accessible, objective indications of project status. Frequent testing and constant assessment of product quality support the practice of frequent releases. Visible test cases and test results can also help emphasize quality as a goal for all projects in the community.

Lint, LCLint, Splint, Checkstyle, JCSC, JDepend, PyCheck, RATS, and Flawfinder The classic Unix command "lint" analyzes C source code for common errors such as unreachable statements, uninitialized variables, or incorrect calls to library functions. More recently designed programming languages have tighter semantics, and modern compilers perform many of these checks during every compilation. LCLint and splint go substantially further than "lint" by analyzing the meaning of the code at a much deeper level. Checkstyle, JCSC, and PyCheck look for stylistic errors in Java and Python code. RATS and flawfinder look specifically for potential security holes.

Adoption and use of these tools is limited, but several projects seem to have started using Checkstyle and JDepend as part of Maven. Analysis tools can also be viewed as a form of reuse where community knowledge is encoded in rules. Relying on standard rules can help open source developers avoid discussions about coding conventions and focus on the added value of the project. Static analysis can prompt peer review and help address weaknesses in the knowledge of individual developers.

Codestriker Codestriker is a tool for remote code reviews. Developers can create review topics, each of which consists of a set source file changes and a list of reviewers. Reviewers then browse the source code and enter comments that are related to specific lines of code. In the end, the review comments are better organized, better contextualized, and more useful than an unstructured e-mail discussion would have been.

Codestriker seems well matched to the open source development practices, but its usage does not seem to be widespread yet. It is accessible to all project members, because it is Web-based and conceptually straightforward. If it were widely used, its model of inviting reviewers would give an important tool to project leaders who seek to turn consumers into volunteers by giving them tasks that demand involvement and prepare them to make further contributions.

Collaborative Development Environments

SourceCast and SourceForge CDEs, such as SourceCast and SourceForge, allow users to easily create new project workspaces. These workspaces provide access to a standard toolset consisting of a web server for project content, mailing lists with archives, an issue tracker, and a revision control system. Access control mechanisms determine the information that each user can see and the operations that he or she can perform. CDEs also define development communities where the same tools and practices are used on every project, and where users can browse and search projects to find reusable components. Both SourceCast and SourceForge include roughly the same basic features. However, SourceCast has been used for many public and private mid-sized communities with an emphasis on security. And SourceForge has demonstrated enormous scalability on the public http://sourceforge.net site and is also available for use on closed networks.

CDEs have been widely adopted by open source projects. In particular, a good fraction of all open source projects are now hosted on http://sourceforge.net. CDEs provide the infrastructure needed for universal access to project information: they are Web-based, and both SourceCast and SourceForge have been internationalized. The use of a standardized toolset helps projects avoid debating tool selection and focus on their particular added value. SourceCast offers a customizable and fine-grained permission system that supports diverse usage in both open source and corporate environments. SourceForge provides specific support for managing the deliverables produced by frequent releases.

Missing Tools

Although there is a wide range of open source software engineering tools available to support many software engineering activities, there are also many traditional development activities that are not well supported. These activities include requirements management, project management, metrics, estimation, scheduling, and test suite design. The lack of tools in some of these areas is understandable, because open source projects do not need to meet deadlines or balance budgets. However, better requirements management and testing tools would certainly seem just as useful in open source work as they are in traditional development.

The Impact of Adopting OSSE Tools

Drawing conclusions about exactly how usage of these tools would affect development inside a particular organization would require specific knowledge about that organization. However, the previous descriptions can suggest changes to look for after adoption:

- Because the tools are free and support casual use, more members of the development team will be able to access and contribute to artifacts in all phases of development. Stronger involvement can lead to better technical understanding, which can increase productivity, improve quality, and smooth hand-offs at key points in the development process.
- Because the "source" to all artifacts is available and up-to-date, there is less wasted effort due to decisions based on outdated information. Working with up-to-date information can reduce rework on downstream artifacts.
- Because causal contributors are supported in the development process, nondeveloper stakeholders, such as management, sales, marketing, and support, should be more able to constructively participate in the project. Stronger involvement by more stakeholders can help quickly refine requirements and better align expectations, which can increase the satisfaction of internal customers.
- Because many of the tools support incremental releases, teams using them should be better able to produce releases early and more often. Early releases help manage project risk and set expectations. Frequent internal releases can have the additional benefit of allowing rapid reaction to changing market demands.
- Because many of the tools aim to reduce unlikable work, more development effort should be freed for forward progress. Productivity increases, faster time-to-market, and increased developer satisfaction are some potential benefits.
- Because peer review is addressed by many of the tools, projects may be able to catch more defects in review or conduct more frequent small reviews in reaction to changes. Peer reviews are generally accepted as an effective complement to testing that can increase product quality, reduce rework, and aid the professional development of team members.
- Because project Web sites, accessible issue trackers, and CDEs provide access to the status and technical details of reusable components, other projects may more readily evaluate and select these components for reuse. Also, HOWTOs, FAQs, mailing lists, and issue trackers help to cost-effectively support reused components. Expected benefits of increased

reuse include faster time-to-market, lower maintenance costs, and improved quality.

▪ Because CDEs help establish communities, they offer both short- and long-term benefits. In the short term, development communities can reduce the administrative and training cost of using powerful tools, and make secure access to diverse development artifacts practical. CDEs can reinforce and compound the effects of individual tools, leading to long-term benefits including accumulation of development knowledge in a durable and accessible form, increased quality and reuse, and more consistent adoption of the organization's chosen methodology.

Notes

1. SourceForge is a trademark of VA Software Corporation.
2. SourceCast is a trademark of CollabNet, Inc.