Supporting Distributed and Decentralized Projects: Drawing Lessons from the Open Source Community

Justin R. Erenkrantz
Univ. of California, Irvine
jerenkra@ics.uci.edu

Richard N. Taylor
Univ. of California, Irvine
taylor@uci.edu

June 2003

ISR Technical Report # UCI-ISR-03-4

Supporting Distributed and Decentralized Projects: Drawing Lessons from the Open Source Community

Justin R. Erenkrantz, Richard N. Taylor
Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3425
{jerenkra,taylor}@ics.uci.edu

ISR Technical Report # UCI-ISR-03-4

June 2003

Abstract: Open source projects are typically organized in a distributed and decentralized manner. These factors strongly determine the processes followed and constrain the types of tools that can be utilized. This paper explores how distribution and decentralization have affected processes and tools in existing open source projects with the goals of summarizing the lessons learned and identifying opportunities for improving both. Issues considered include decision-making, accountability, communication, awareness, rationale, managing source code, testing, and release management.

Supporting Distributed and Decentralized Projects: Drawing Lessons from the Open Source Community

Justin R. Erenkrantz, Richard N. Taylor

Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3425
{jerenkra,taylor}@ics.uci.edu

ISR Technical Report # UCI-ISR-03-4, June 2003

Abstract

Open source projects are typically organized in a distributed and decentralized manner. These factors strongly determine the processes followed and constrain the types of tools that can be utilized. This paper explores how distribution and decentralization have affected processes and tools in existing open source projects with the goals of summarizing the lessons learned and identifying opportunities for improving both. Issues considered include decision-making, accountability, communication, awareness, rationale, managing source code, testing, and release management.

1. Introduction

Organizational distribution and decentralization alter critical factors in the software development process. Historically, centralized organizational structures have prevailed. A single organization, or part of an organization, has been fully responsible for a project, bearing the ultimate responsibility for shaping the deliverables and selecting the tools and processes used in development. Communication and consensus building within the organization has been facilitated by physical proximity.

For numerous business reasons, over the past decade and more, many organizations have moved to *distributed* development — a single administrative authority operating over physically distributed subgroups. This change has been supported by improvements in communication and networking technologies. Nonetheless, with participants no longer physically collocated, the processes and tools of development have had to change to attempt to cope with the difficulties so incurred.

Development of applications by *decentralized* organizations adds an additional wrinkle into the problem. By decentralized development we mean that no single organization controls the project; rather that all decisions related to the goals and objectives for the project must be made multilaterally. The motivation for decentralized development is akin to the motivation for participation in standards bodies: the common weal can be advanced while independence is retained. Each organization holds ultimate authority for its internal processes and tools. To the extent that interaction between participating organizations occurs, selection of the involved tools and processes must also be done multilaterally.

The premise of this paper is that we can gain some insight into how to effectively meet the challenges that face decentralized and distributed development organizations by examining the practices of the open source community, as these projects are most often both distributed and decentralized. The intended beneficiaries of this work is, largely, new open source projects, through several of the observations have applicability outside the open source domain.

Some work along this line has already taken place. Progressive open source[6], for instance, has been introduced in some commercial entities. This model is primarily geared towards applying open source practices within the community of internal employees or specific strategic partners rather than the public. This model is not fully decentralized, however. Implicit in the notion of progressive open source is a controlling authority that can dictate development.

There has also been a large body of work related to distributed software development. One particular area that has been carefully studied by Herbsleb, *et.al.* is the communication between participants in a distributed software project[15]. In this case study, participants were spread across several countries and developed a project collaboratively. One of the significant results was that there was a clear bias towards communicating with people in proximity, rather than communicating with remote peers, even when supported by good communication technology. This study does not fully explore the effects of decentralization on development, however, as all of the participants essentially worked for the same organization and were clustered in relatively large groups at a small number of sites. In a highly decentralized and distributed software project, few developers may be in proximity and belong to the same organization.

Another body of work has focused on enhancing technologies specifically for supporting distributed development. CSCW technologies fit into this category, as well as enhancements to the web. In [7], for instance, enhancements were discussed that could make the current web tools better facilitate collaboration. However, it limited itself to web-based artifacts and does not lay out a guideline for the processes best suited to these tools.

It almost goes without saying that not all projects require heavyweight processes and tools due to their limited scope or participation. Introducing unnecessary processes and tools may stifle a small project. It is also possible that participants do not desire expansion beyond a specific threshold. These classes of projects do not truly fit the distributed and decentralized criteria. In the following discussion, therefore, we will only concern ourselves with projects that are sizeable or complex enough to warrant tool and process support and which are developed in a collaborative, distributed, and decentralized fashion.

We begin the remainder of the paper with discussion of a survey of open source projects, showing similarities that have arisen in tool usage. Discussion then turns to characterizing the ways distribution and decentralization can constrain processes and tools. We then begin to summarize lessons from the open source experience, starting with identifying the management and coordination needs. We continue with an examination of techniques for satisfying these various needs. We conclude with a discussion of potential future work.

2. Basis Projects

Since most open source projects display significant degrees of distribution and decentralization in their organization, they provide a good foundation for study. Some prior examinations have been conducted into the tool usage of open source projects[12]. While most open source projects are not directly related to each other in terms of the subject of their production, a commonality of supporting tools has emerged in many cases.

In [12], eleven open source projects were surveyed to determine what tools are used to support the development model of the project. The survey was conducted to determine the quality aspects of open source projects and determine how to improve the project deliverables. The surveyed projects are spread across several different domains - including compilers, web servers, programming languages, and desktop environments.

The surveyed projects are among the most successful open source projects available. The Apache HTTP Server is currently in use by about sixty percent of all websites[20]. Servers shipping with the Linux kernel amounted for fourteen percent of all servers shipped in the first quarter of 2003[11]. Tomcat is the official reference implementation for the Java Servlet and JavaServer Pages technologies[2]. Therefore, these projects provide a reasonable basis for examining how successful distributed and decentralized open source projects should be conducted.

As Figure 1 depicts, these projects also represent a wide range of source code size. One project had as little as 55 thousand lines of code (Tomcat), while another surveyed project supports 2.5 million lines of code (Linux).

Each project has independently chosen the tools and processes that best fit it. Most of the surveyed projects do not have any common developers, so there is no direct relationship. However, in some areas, a consensus appears to have been reached concerning the proper tools to use.

In the survey, all of the projects shared the same source control system, CVS. However, since the publication of the survey, Linux has adopted BitKeeper as its source control system[4]. While there does currently appear to be a consensus regarding CVS, a number of other replacements to CVS are actively being developed. These include such

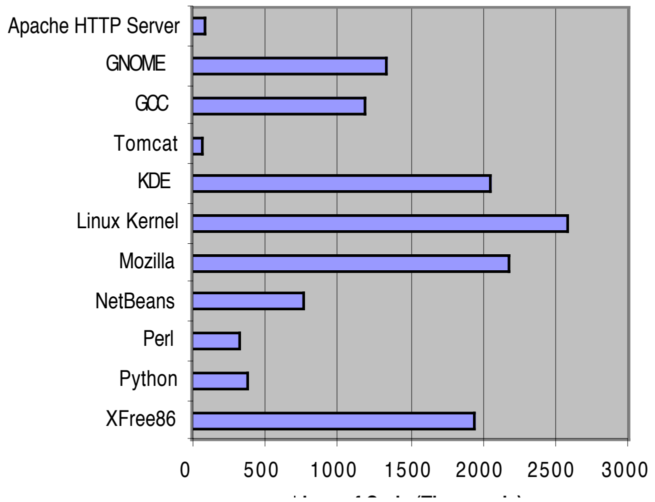

Figure 1: Lines of Code in Basis Open Source Projects

tools as Arch[1] and Subversion[5]. Therefore, this consensus may not be stable over the long-term as newer products attempt to replace CVS.

In other areas, there is no single tool that predominates; rather, a small number of tools are commonly used. One such area is in mailing list software; two tools currently dominate - ezmlm[3] and Mailman[10]. While two of the surveyed projects used other tools, the rest used one of these two tools.

In yet other areas, such as web portals, there is extreme variations in the tools used. No two projects shared the same tool for updating their website. At this point, most projects are creating customized tools for their website that fit their individual needs rather than relying upon a prebuilt solution for content management.

The variations of tool similarity across problem domains presents an interesting statement. In some areas, open source projects have found a particular tool that seemingly fits their development model well. This is evidenced by consensus concerning a particular tool. This consensus may be due to an inherent property of the way the project is organized whereby this tool is the only obvious choice. Or, perhaps the adoption of a particular tool is a matter of historical accident. If the adoption is related to historical accident rather than a solid fit, the introduction of tools that are better suited to a distributed and decentralized development model may be able to replace the current consensus. However, in order to encourage better tools and processes, we must understand the constraints placed upon an open source project by decentralization and distribution.

3. Constraints

The presence of decentralization and distribution in a software project places a number of new constraints on what processes and tools can be effectively utilized. In order to obtain a clearer picture of what may work in these environments, we need to be able to identify these constraints.

3.1. Decentralization

The decentralization aspect of development requires processes to consider multiple interested parties. The involved developers may act towards their own goals, rather than the goals of the entire project. Therefore, not all developers will necessarily be aligned on all items and tasks. Yet, the processes and tools used should try to promote working towards a common beneficial goal while meeting the individual goals.

Due to decentralization, developers may not all work for the same physical organization. However, one organization may fund a portion of the developers on a project. If this organization removes its funding, their associated developers may leave the project. Therefore, the project needs to be able to withstand such losses or risk having the project abandoned. This risk promotes processes and tools which maintain continuity and shared communal knowledge.

When the individual goals of organizations collide, care should be taken in resolving these concerns. If these concerns are not met to everyone's satisfaction, dissatisfied organizations may leave the project. Depending upon the influence of the departing subset, it may place the project in jeopardy. Therefore, processes promoting compromises should be strongly emphasized to minimize such departures.

3.2. Distribution

Distributed software development places a strain on the project's communication mechanisms. When developers are not collocated, it is no longer possible to have frequent face-to-face meetings. Therefore, other communication mechanisms and tools must be deployed to fill this void.

As noted earlier, prior studies into the nature of distributed software development have indicated that it is hard to facilitate communication to the right person at the right time across site boundaries[15]. In order to address this problem, processes and tools need to be in place to allow timely identification of key contacts. Herbsleb, *et.al.*, for instance, identifies needs in the areas of awareness, rich interpersonal interaction, and support for finding experts.

Since developers are not physically collocated, it may cause problems with synchronous communication as developers may be scattered across timezones. If synchronous methods are used, some participants may not be able to contribute to a discussion. Therefore, asynchronous communication mechanisms are usually preferred.

4. Management and Coordination Needs

This section identifies management and coordination needs that decentralized and distributed project organization imposed. If these needs are not properly addressed at the outset, then repercussions may arise as the project matures.

4.1. Goals

Before embarking on a project, there is usually a need for a clear statement of goals that the project needs to accomplish in order to be successful. Upfront identification of goals allows for examination by prospective participants. It may be that the initial goals may not suit all interested parties. Therefore, they may need to be altered to support other interests. A consensus about the end result can then be built by the new participants. If the interests are thus made to correspond, the groups can begin to proceed to coordinate development tasks. If their interests are irreconcilable, the parties may proceed separately. A confrontation may occur later if an implicit differences in goals is later revealed.

Furthermore, if only a few parties wish to participate, the potential cost of decentralization may not add sufficient value to the project. It may be that the project does not have enough outside attraction to reach a critical mass to support a viable community. Unnecessarily adding the overhead of decentralization may end up harming the viability of the project.

4.2. Coordination of Initial Development

Once a goal has been determined, the interested parties need to identify how to reach these objectives. This roadmap can be valuable in planning development activities. A project may have an initial donation of code to build upon, or the new project needs to start the development process.

Inherited code

A project may inherit code based upon a prior effort that has decided not to further development, or, one of the interested parties may be willing to donate code to begin the development effort. In either case, interested parties should be aware of the implications of the decision.

When using inherited code, the primary task becomes enhancement and evolution. At first, the project may be able to bypass the design stage of the software life-cycle. The majority of tools and processes will be geared towards maintenance. Depending upon completeness of the donation, design artifacts may need to be reproduced to promote understanding of the inherited code.

As the project matures, limitations may be found in the initial design that require substantial refactoring. The initial developers may desire a reasonable expectation that the inherited code allows for ample extensibility. Otherwise, efforts to evolve the code may encounter an early roadblock that forces reconsidering the usage of this code.

Initial code

When a project begins afresh, the initial processes and tools will primarily be design-oriented rather than maintenance-oriented. In order to work in a distributed environment, the processes and tools must be able to support collaborative design. As the project evolves, the processes may alter to primarily supporting implementation and maintenance tasks.

A common occurrence in open source projects is that a publicly documented standard is implemented. These documents are typically written by a separate standards organization. These documents serve as the initial requirements and often specify interoperability characteristics of the deliverable. Once agreeing upon a standard to implement, developers can then devise a plan to carry out the architectural design and implementation.

By minimizing the requirements stage of the software life cycle by leveraging pre-existing standards, more effort can be directed towards the design and implementation. However, many projects implementing public standards also provide feedback to the standards committee based on real-world implementation experience.

Effect on design and requirements

Since some of the most prominent open source projects inherited code which implements a public or well-known standard, it may stand to reason that the processes involved with design and requirements gathering are not as well developed as maintenance and extensibility of code in open source projects.

However, in these particular cases, the requirements and initial design have already been determined in a very rigorous manner. In the case of projects which implement Internet RFCs, these requirements have been created in a very decentralized fashion. However, once these requirements have been established, various parties will form groups to implement the standard.

Therefore, while it may seem that some open source projects lack an emphasis on requirements and design, we may be able to rationalize that on the strict division of requirements and implementation in the traditional standard-making bodies of the Internet.

4.3. Common procedures

In decentralized communities, the interested parties may establish a common set of rules for running the project. These rules will take the place of a controlling authority which dictates such rules. Furthermore, this will allow all parties to operate within stated organizational parameters.

The parties should have already agreed on the goals and may have agreed on the initial design, but they must now also agree how to reach the desired result in a formal manner. If a conflict between members of the project arises, there needs to be a predetermined mechanism for resolving these conflicts.

If these steps are ignored and such a process does not exist, it may introduce tension between parties. By creating and following these guidelines, the belief is that most conflicts will be resolved peacefully. These procedures should also attempt to not introduce unnecessary overhead in the development process.

If these mechanisms fail and an impasse develops, then the community may be *forked*. One of the most prominent examples of the forks of open source projects are among the BSD-derivatives[17]. Despite having a common ancestry, the vision of each BSD-derivative is slightly different. In essence, each BSD-based platform has the goal of creating a Unix-like operating system. However, each of these derivatives has a different technical or procedural vision of how this goal should be accomplished.

Therefore, we suggest that decentralized projects are self-correcting though at a cost of wasted resources. When a difference of vision occurs between developers and organizations, projects can be forked to maintain the integrity of the private goals. In the end, each constituency retains their private goals, and may be willing to separate from other participants if an impasse is reached. Only when their private goals are being met will participation continue.

4.4. Tool requirements

When multiple parties participate in decentralized development, special attention should be made to the requirements of the tools that support the processes. The selection of tools should recognize that not all developers have equal resources to acquire specialized tools. Open source projects may not be directly funded, but when all participants are funded, these requirements may not be as stringent.

Since open source projects are traditionally open to all developers regardless of organizational affiliation, the tools used are commonly open source as well. By relying on free tools, this alleviates financial barriers to participation as not all developers may receive direct compensation for their work on a project. It may be unreasonable to expect developers to purchase tools in order to work on a project.

Due to the variety of developer preferences, most required tools need cross-platform support. In the open source community, a good tool will not require developers to switch their operating system to use a special tool. This allows developers to work on platforms with which they are most comfortable.

Since a project may attract developers of different skillsets, it may be unreasonable to expect developers to have special training in a tool or a technique. To offset this, projects may need to provide clear documentation on techniques that will help unfamiliar developers. Furthermore, since the participants are self-selecting, not all participants may have formal computer-science backgrounds, so some more advanced techniques may not be accessible to all participants.

5. Process and Management Tools and Techniques

This section describes process and management techniques that may be used in distributed and decentralized projects. These techniques attempt to satisfy the needs discussed previously. We will examine how open source projects are solving these constraints and identify potential areas of improvements for each concern. Table 1, at the end of the paper, will summarize these techniques.

5.1. Delegation and Decision-Making

A concern for distributed and decentralized projects is delegation of assignments and leadership. Since the participants do not necessarily share the same reporting structure, traditional management techniques may not apply.

Similarly to other management models, there may either be a flat or hierarchical structure within this decentralized organization. Ultimate authority may rest with a specific individual, or decision-making responsibility may be shared by the interested participants. In the case of a single authority, this individual may set policies unilaterally. However, these policies must still promote participation by others. This requires the creation of a benevolent dictatorship where participants are willing to yield authority to a central authority — explicitly backing away from decentralization.

A prominent example of this central authority organizational model is seen in the Linux kernel. Linus Torvalds was the initial designer and developer of the Linux kernel. The rest of the participants have allowed him to maintain control over the project. Linus has the ultimate say on what changes make it into the kernel.

Designating a single individual with ultimate authority may create an organizational bottleneck. Therefore, a hierarchical organizational structure usually accompanies these structures. In Linux, most substantial components of the kernel have an associated maintainer. Rather than submitting a change directly to Linus, changes should be submitted to the responsible maintainer. If the maintainer agrees with the patches, the patches can then be submitted to Linus.

Linus places a certain degree of trust in his maintainers that they will follow his process for submitting patches to him and deal with most of the overhead for that component Yet, due to the supreme nature of Linus's role, he can overrule the maintainer of a component. It is possible to circumvent a maintainer and send a patch directly to Linus. If he decides to apply the changes without receiving prior input from the maintainer, he retains that right.

Another model commonly used by open source projects, one more in tune with decentralization, is the meritocracy model. This is exemplified by the Apache HTTP Server Project[9]. All members share equal power, so there is no direct leader of the project. Under this flat organizational model, people gain power by sustained contributions over time. The power of the developer is enabled by grants of write access to the shared repository and the ability to veto changes.

Until a developer gains commit access, they are considered a contributor. While they may participate freely on the mailing lists, an intermediary with appropriate access must review and commit their suggested changes. They may also cast non-binding votes on issues before the community. Since these votes are non-binding, developers with binding votes may choose to disregard such votes.

As the voting developers are exposed to a new participant, they are examining the quality of the contributions and how the participant works within the community. Then, one voting developer will nominate the contributor for voting privileges to the rest of the voting developers on a private discussion list. If the group considers the contributions beneficial and the participant trustworthy, voting privileges will be offered.

While no single person can control the project, each voting developer has *veto* authority to stop undesired changes from being merged into the shared repository. While these vetoes can be cast on any patch, there must be a valid technical reason for stopping this change. There is also no way to override a veto - this organizational model enforces consensus-building.

5.2. Accountability

Accountability may become an issue in a decentralized organization. If there is a problem with the software, users may desire a contact to resolve this problem. Open source projects have typically addressed this concern in two fashions: creating for-profit corporations that provide commercial support or creating non-profit foundations that provide a perpetual point of contact. These solve the issue by a direct step away from decentralization.

Some organizations that participate in open source projects provide for-fee support as a source of revenue. For example, this is common in the relational database domain. Two open source databases, PostgreSQL and MySQL, both have strongly related corporations that sell support to end-users.

These commercial entities will often provide support plans that assist users in setting up the product. These companies may also respond to direct support questions concerning the product. By having a revenue stream, these companies are able to fund development of the associated open source project by directly financing developers. These developers may add enhancements that the organization's client base has requested, or fix problems that have been identified by support personnel.

As an alternative to providing a commercial support, some open source projects have established non-profit foundations. These foundations are the owner of the code and do not have any explicit commercial interests. Therefore, it is expected that these foundations will be able to oversee and maintain accountability for the code. Two prominent examples of this are the Free Software Foundation and FreeBSD Foundation.

In these cases, a non-profit foundation is usually responsible for providing the infrastructure of the project. They will typically provide the services that allow development to occur. These foundations do not usually provide end-user support or directly fund developers. However, the foundation is expected to be eternal, while a for-profit corporation may be forced to dissolve due to financial considerations.

5.3. Communication

Due to the introduction of distribution, there may be varying degrees of developer collocation. Since the projects are also typically decentralized, developers may not work for the same physical organization. Therefore, the development process must allow for communication between people not at the same location and not belonging to the same physical organization. Therefore, the majority of communication should be at the virtual organization level, rather than the physical organization. By relying upon asynchronous forms of communication rather than synchronous communications, a higher proportion of global developers can be supported. Yet, relying upon asynchronous communication introduces a delay factor[8,15].

In order to facilitate communication to the right person at the right time, mailing lists are commonly used. This reduces the number of contacts that are required. Almost every open source projects uses public mailing lists to promote subscription by non-developers and to encourage contributions by new developers.

Multiple mailing lists may also be used to further segment the mail traffic. These mailing lists may be dedicated to a particular sub-topic. By reducing the scope of a mailing list, it allows for separate communities to form within the same project. This may be beneficial for encouraging growth in large projects. It also moves discussion away from a more generic mailing list where there may not be as many interested people in the discussion.

It has been stated that email is predominately used because it is the least common denominator[8]. One problem with email is that it requires a common language to be used. Mechanical translation services have not yet proved to be sufficient to address technical translations. This may promote developers who are only fluent in the main language of the developers.

A possible avenue to research would be to investigate projects where developers do not share a common language. In these cases, it would be useful to analyze how developers communicate when they do not share a language. This may promote islands of developers that do not often communicate.

In addition to relying upon asynchronous communication, some projects use a variety of synchronous communication (such as real-time chats or instant messaging). Yet, this is only effective when developers are located in similar time zones. If not all developers can participate in synchronous communications, it is essential that some archival of the communications be made. Otherwise, key participants may be left out of a critical discussion.

5.4. Awareness

Awareness is an understanding and coordination of what participants are doing. Since the personnel of a decentralized and distributed community may be rapidly changing, it may be hard to even identify who is currently active. This aspect of development processes has been remarkably underdeveloped. Most coordination efforts remain ad-hoc and short-term.

However, mailing lists provide a rudimentary tool for coordination. A developer can post on the mailing list indicating that they are planning to perform some activity. But, there is no enforcement of this plan. This leads to a problem when a participant says they are going to accomplish some task, but does not complete it.

Some projects may also use shared information repositories for awareness information. For example, the Apache HTTP Server Project relies on a STATUS text file that lists outstanding issues; this file is emailed weekly to the main developers mailing list. Participants may make a notation as to which issues they are addressing. However, it may require frequent refreshing of this file to ensure that the information is not stale.

Many open source projects also require that large changes be discussed before implementation starts. This allows other developers to provide feedback on proposed implementation strategy. Leveraging the feedback of developers may allow potential design problems to be detected earlier than if review occurs after implementation.

5.5. Historical Rationale

Since turnover may be high in decentralized projects, a collective history should be maintained and documented. By examining past communications and activities, new developers can begin to understand decisions made at a certain point in the past. It also allows developers to learn from prior decisions.

It is essential to use communication mechanisms that allow for long-term archival. Most asynchronous forms of communication lend themselves well to archiving - such as public mailing list archives. Therefore, the delay factor introduced by asynchronous communication has an advantage of allowing capture of historical rationale.

However, spontaneous synchronous communications are often not archived. This may often be seen in projects where a number of developers are physically co-located. In these environments, face-to-face communications may have an unusually strong bias[15]. In addition to not allowing full participation of the group, these sorts of communication may be detrimental to distributed projects because artifacts of these conversations are rarely recorded.

A common problem in open source projects is that new developers often repeat or bring up old discussions. This demonstrates a lack of tools that encourage review of past discussions. If tools for reviewing prior discussions were readily available, developer time spent rehashing prior topics could be minimized.

To help with this, Perl has created Perl Design Documents (PDDs) which lay out the rationale for certain decisions made in the development of Perl 6 and Parrot[28], This allows new developers to annotate and reexamine prior decisions in a central location. It may happen that a new developer has added insight that was not noted in the prior conversation. If post-mortem annotation of discussions is allowed, it may achieve a balance between stifling and encouraging reexaminations.

5.6. Design Rationale

In addition to allowing for discovery of important historical conversations, it may be critical to understand the design rationale of certain components. In a distributed and decentralized environment, it may not be possible to contact the original author of a section of code. Therefore, mechanisms need to be in place to communicate rationale to future participants.

One way to communicate rationale is by creating developer documentation. Some open source projects, such as AbiWord[25], keep interface definitions and notes in-line with the source code. Documentation can then be published with tools such as doxygen[14]. By synchronizing the location, when major changes are made to the source code, the belief is that the documentation will be updated. This makes it easier to produce developer documentation which reflects the current code.

Depending upon whether the project did the initial design, other artifacts such as design documents and diagrams may be available. In projects that provide an extensible interface, it is also common to produce well-explained and concise examples as a way to illustrate the interface in action. This helps new developers of an interface understand the code by looking at examples.

There has been research into encouraging software reuse, but these tools have not yet been integrated into the mainstream. There are tools available that provide relevant interface information in a personalized manner[30]. There has also been work towards harvesting the structural and semantic information of code[19]. Encouraging adoption of already existing tools may make capturing design rationale easier.

5.7. Participation

In projects where the personnel on a project may change frequently, it is important to have a published set of developer guidelines. These guidelines allow familiarization with the processes and tools used in a project. New participants can review them and contribute to the project in an intelligent manner.

Sites such as the KDE Developer's Corner provide a wealth of information that allow new participants interested in KDE to learn how to contribute[18]. The site contains introductory tutorials for developers new to the internals of KDE. Information about the development tools required to compile KDE and how to obtain the latest KDE snapshots are also available.

It has been mentioned that having an established set of guidelines shared by projects can reduce the redeployment costs of developers[6]. If all projects shared the essential guidelines, it would make it easier to contribute to new projects. If each project had its own set of unique guidelines, it would be difficult to transition to new projects. Therefore, it would be beneficial to encourage standardization of participation guidelines across projects.

5.8. Controlling Participation

A corollary to encouraging participation in decentralized and distributed software is that participation by new people must be managed by the current participants. Depending upon the access policies of the project, new participants may only have limited access to making changes to the project. Therefore, processes and tools need to be in place to support facilitating such contributions.

Tools such as the SourceForge's patch manager used by the Python project can be extremely useful[21]. These tools allows participants to submit patches to be applied, then developers with the appropriate commit access can integrate the changes. This particular tool also allows for annotations to be stored.

However, these current tools suffer from a lack of integration with the rest of the development process. Some projects enforce a policy where a certain number of positive reviews must be received before a change can be integrated[26]. Contributions may also grow stale as the project code base evolves. Furthermore, if none of the active developers deem an issue important, it may be a challenge to motivate integration. The tools used to control participation should ease the burden of merging the changes.

5.9. Managing Source

Since the developers are distributed, it is often a requirement to have a unified view of the source code. If a unified view is not available, it may be possible for developers to not be aware of the current state of affairs. Therefore, most projects will adopt some sort of collaborative software configuration management system (SCM). The processes and tools need to balance that each developer should be able to work independently while allowing them to remain consistent with the rest of the team.

As discussed previously in [12], the predominate SCM in use by open source projects is currently CVS. There has been a recent trend in seeking tools that can replace CVS[1, 4, 5]. CVS is based on a centralized repository model with one repository holding all of the content. Some of the newer SCM tools that have been introduced are keeping the centralized model of CVS[5], while others are attempting to decentralize the repository structure[1, 4].

However, depending upon the accountability structure of the project, it may make sense to keep a centralized repository even in a decentralized project. If a project has a non-profit organization which holds the copyright, then this organization should administer the master repository. However, if the project has a loose accountability structure, a decentralized repository structure may be more efficient.

Most SCM tools currently in use also promote an optimistic conflict resolution model rather than a pessimistic conflict resolution mode[16]. An optimistic locking strategy allows source conflicts to be resolved at commit-time, while a pessimistic locking strategy uses locking to prevent others from making changes while a change is being developed. A pessimistic locking strategy may interfere with parallel development as it prevents two developers from working on the same file at the same time. Only using pessimistic locking may have an impact upon the effectiveness of distribution.

5.10. Issue Tracking

One of the stated advantages of open source projects is that it is easier to fix problems since the source code is freely available[22]. However, it may still be difficult for non-developers to fix problems as they may not have the appropriate background required to resolve a defect. Therefore, processes and tools are required to report problems to the people who can help resolve defects.

Due to the presence of decentralization, it may be difficult to solicit participants who can resolve reported defects in a timely manner. Some participants may be wary of working with end-users, or are too busy to deal with acquiring the relevant information from the reporter. Therefore, the tools need to be able to support novice users and expert developers efficiently.

Standardizing on issue tracking tools, such as Mozilla's Bugzilla[27], may increase the familiarity of both users and developers with these tools. However, as these tools are adopted by more projects and enhancements requested, feature creep must be resisted. If the issue tracking tool becomes too complicated to use effectively, its usefulness is diminished.

5.11. Documentation

Since not all users of a project are developers that can understand code, a project must also be able to deliver quality user documentation. Otherwise, users may find the product too complicated to use properly. A significant challenge for distributed and decentralized projects is to have documentation at an equivalent quality to the code.

At best, documentation can be viewed as a form of source code. Therefore, many of the processes that apply to source code can also apply to documentation. Documentation may be written in a collaborative environment using similar tools and processes as the ones used to write code.

A problem in any software project is how to keep the end-user documentation synchronized with the current version of the source code. Oftentimes, developers are hesitant or reluctant to write user documentation. Therefore, when they make a change that is visible to a user, the developer may not update the relevant documentation. This leads to the documentation becoming out of sync with the code.

Some open source projects have addressed this by having separate documentation teams. One example of this separation is in PHP's documentation[29]. By isolating the documentation process from the development process, it enforces another perspective on the usability aspects of the code. This may result in an increase of quality of both the code and end-user documentation.

Another characteristic of the PHP documentation process is that it allows users to annotate the documentation on the website. As users spot errors in the documentation, they may append a correction comment to the website. Then, PHP documentation participants can harvest the changes into the main documentation.

5.12. Testing

There is often a strong desire to ensure that a project meets both the functional and reliability goals previously established. Therefore, testing processes and tools should be developed and encouraged throughout the life cycle of the project. There are two classes of methods that are typically used in open source projects: code review and testing.

Since it is difficult to conduct regular meetings in a distributed workplace, it is not possible to conduct periodic code review sessions. Therefore, code reviews must occur as the changes are conducted. Developers are usually asked to make small verifiable changes rather than large changes. By asking all developers on a project to review the changes as they happen and asking for the most concise changes possible, it may make it easier to identify problems sooner.

Besides relying upon manual inspection, some projects have a suite of automated tests for the project. These automated tools allow all participants to run the same set of tests at their discretion on their specific platform. One such project that utilizes automated tests is Subversion[5]. The test suite in Subversion is extensive and tests almost all functionality of the system. Furthermore, no releases can be made without first passing the automated tests. It may be possible to integrate some recent research into optimizing which regression tests are executed to improve the performance of the test suites[13].

5.13. Release Management

Since users ostensibly wish to use the deliverables of a project, quality releases must be produced. Therefore, a viable release strategy must be determined. If the project does not have a coherent process in place, it may have problems attracting users or achieving a reputation for stability.

In order to achieve widespread distribution, an infrastructure must be in place to allow public consumption. Some projects rely upon mirrored servers to handle the load of delivering releases to end-users. A critical concern is how to select these mirrors - should they be self-selected or should they be limited only to trusted individuals.

One such project that relies upon mirrors to deliver releases is Debian[23]. Debian balances the load across

Table 1: Summary and Avenues for Enhancements

Issue	Techniques	Project Exemplar	Avenues for Enhancements
Decision-Making	Project leader, meritocracy	Linux, Apache	Understanding consequences
Accountability	For-profit support, non-profit ownership	PostgreSQL, FreeBSD	Introducing clarity
Communication	Discussion lists, asynchronous	All	Balancing granularity
Awareness	Frequent status updates, Discussion before implementation	Apache	Creating better tools
Historical Rationale	Archival of communications, design documents	Perl	Creating better tools
Design Rationale	Developer-centric docs, examples	AbiWord	Enforcing synchronization
Participation	Clear tutorials, guidelines	KDE	Creating standards
Controlling Part.	Feedback, annotating contributions	Python	Integrating into other processes
Source Code	Public source repository, optimistic conflict resolution	All	Investigating decentralization
Issue Tracking	Soliciting developer assistance	Mozilla	Creating easy-to-use tools
Documentation	Distinct personnel, annotations	PHP	Separation of code and docs
Testing	Code reviews, automated tests	Subversion	Optimizing test executions
Release Management	Mirroring, versioning	Debian	Managing distributions

many geographically dispersed self-selected servers. However, Debian has several push-primary mirrors that are chosen because of their reliability. Self-selected mirrors can then pull releases from one of the pushed mirrors rather than accessing the master site directly.

Projects may also place meanings on the versions that deliverables are labeled with. This allows a shared understanding of the expected reliability. At times, it is helpful for a project to have a development branch that is not intended for widespread usage. These releases can also be used to perform dry-runs of the release process. This can be especially helpful when a project is trying a new release process. By explicitly labeling a version as unstable or development, it can help match the expectations of users with the expectation of the developers.

For example, Debian always has at least three versions that are actively maintained: stable, testing, and unstable[24]. The stable distribution is the one that is recommended for widespread usage. Then, the testing distribution consists of packages that are waiting to be included in the next stable release. Then, the unstable distribution is meant for developers and not meant for production quality.

6. Summary and Future Work

Adopting a decentralized and distributed organization for developing software requires rethinking fundamental process and tools. We have attempted to examine the consequences of supporting decentralization and distribution by seeing how open source projects have addressed these concerns. Table 1 provides a summary of the issues, techniques, and projects discussed in this paper. It also lists avenues for enhancement that have been identified where the current processes and tools could be improved to better support distributed and decentralized software projects.

If projects can create a clear line of accountability that is separate from all of the participants, it may foster a sense in the users that responsibility will be maintained. A decentralized project should be able to withstand the departure of organizations gracefully. If this is not present, then users may become wary of the project falling out of active maintainership.

By limiting the scope of discussion lists, it makes it easier for participants to understand what is currently going on in areas of the project. This level of granularity must be balanced with having too many mailing lists that makes it difficult to find the appropriate forum for discussion. However, when the right balance is achieved, this allows participants to easily partition discussion based upon agreed topical lines.

One concern for distributed software development is that a set of standards is required in order to ease participants shifting from one project to another. This may manifest itself as a common vocabulary shared between projects. If participants do not share a common language, it becomes hard to communicate effectively. The creation of standards and accepted best practices can help ease migration between projects.

A common problem in a distributed software project is understanding what other participants are currently working on. The creation of tools to promote awareness between developers can address this need. Furthermore, tools that promote capturing of historical rationale may make it easier for new participants to enter a project.

Another area for tool improvement is introducing a way to capture the rationale for a decision in the documentation. Currently, it is hard to identify why a particular change is made. The artifacts for determining this may not be centralized. Creating a tool that indicates relationships between artifacts to encourage rationale understanding may be critical.

The current tools for controlling participation are ad hoc and not well integrated. This makes it difficult to lower the burden upon the participants in a project in dealing with the contributions by casual participants. If the tools for handling contributions were better integrated into the standard processes, it would make this task significantly easier.

7. Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 0205724.

8. References

[1] *Arch - Revision Control System*. <http://arch.fifthvision.net/>, HTML, 2003.

[2] Apache Software Foundation. *The Jakarta Site - Apache Tomcat*. <http://jakarta.apache.org/tomcat/>, HTML, 2003.

[3] Bernstein, D.J. *ezmlm*. <http://cr.yp.to/ezmlm.html>, HTML, 2000.

[4] Bitmover. *BitKeeper*. <http://www.bitkeeper.com/>, HTML, 2003.

[5] CollabNet. *Subversion*. <http://subversion.tigris.org/>, HTML, 2003.

[6] Dinkelacker, J., Garg, P.K., Miller, R., and Nelson, D. Progressive Open Source. In *Proceedings of the International Conference on Software Engineering (ICSE)*. p. 177-184, 2002.

[7] Fielding, R., Whitehead, E.J., Anderson, K., Oreizy, P., Bolcer, G.A., and Taylor, R.N. Web-based Development of Complex Information Products. *Communications of the ACM*. 41(8), p. 84-92, 1998.

[8] Fielding, R.T. and Kaiser, G. The Apache HTTP Server Project. *IEEE Internet Computing*. 1(4), p. 88-90, 1997.

[9] Fielding, R.T. Shared Leadership in the Apache Project. *Communications of the ACM*. 42(4), p. 42-43, 1999.

[10] Free Software Foundation. *Mailman*. <http://www.list.org/>, HTML, 2003.

[11] Fried, I. Sales Increase for U.S. Linux Servers. *CNet News.com*. February 10, 2003. <http://news.com.com/2100-1001-984010.html>.

[12] Halloran, T.J. and Scherlis, W.L. High Quality and Open Source Software Practices. In *Proceedings of the Meeting Challenges and Surviving Success: 2nd Workshop on Open Source Software Engineering*. May, 2002.

[13] Harrold, M.J., Jones, J.A., Li, T., Liang, D., Orso, A., Pennings, M., Sinha, S., Spoon, S.A., and Gujarathi, A. Regression Test Selection for Java Software. In *Proceedings of the ACM Conference on OO Programming, Systems, Languages, and Applications (OOPSLA 2001)*. p. 312-326, Tampa, Florida, October, 2001.

[14] Heesch, D.v. *Doxygen*. <http://www.doxygen.org/>, HTML, 2003.

[15] Herbsleb, J.D., Mockus, A., Finholt, T.A., and Grinter, R.E. An Empirical Study of Global Software Development: Distance and Speed. In *Proceedings of the International Conference on Software Engineering (ICSE)*. p. 81-90, 2001.

[16] Hoek, A.v.d. Configuration Management and Open Source Projects. In *Proceedings of the 3rd International Workshop on Software Engineering over the Internet*. Limerick, Ireland, June 6, 2000.

[17] Howard, J. The BSD Family Tree. *Daemon News*. April, 2001. <http://www.daemonnews.org/200104/bsd_family.html>.

[18] KDE e.V. *Developer's Corner*. <http://developer.kde.org/>, HTML, 2003.

[19] Maletic, J.I. and Marcus, A. Supporting Program Comprehension Using Semantic and Structural Information. In *Proceedings of the 23rd International Conference on Software Engineering*. p. 103-112, Toronto, Ontario, Canada, May, 2001.

[20] Netcraft. *Netcraft Web Server Survey*. <http://www.netcraft.com/survey/>, HTML, 2003.

[21] Python Software Foundation. *Patch Manager*. <http://sourceforge.net/tracker/?group_id=5470>, HTML, 2003.

[22] Raymond, E.S. *The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly, 2001.

[23] Software in the Public Interest. *Debian Mirrors*. <http://www.debian.org/mirror/>, HTML, 2003.

[24] Software in the Public Interest. *Debian Releases*. <http://www.debian.org/releases/>, HTML, 2003.

[25] SourceGear Corporation. *AbiWord Documentation*. <http://www.abisource.com/doxygen/>, HTML, 2003.

[26] The Apache HTTP Server Project. Apache HTTP Server Project Guidelines and Voting Rules. <http://httpd.apache.org/dev/guidelines.html>, HTML, 2003.

[27] The Mozilla Organization. *Bugzilla Project Homepage*. <http://www.bugzilla.org/>, HTML, 2003.

[28] The Perl Foundation. *Parrot and Perl6 PDDs*. <http://dev.perl.org/perl6/pdd/>, HTML, 2003.

[29] The PHP Group. *PHP: Documentation*. <http://www.php.net/docs.php>, HTML, 2003.

[30] Ye, Y. and Fischer, G. Supporting Reuse by Delivering Task-Relevant and Personalized Information. In *Proceedings of the 24th International Conference on Software Engineering*. p. 513-523, Orlando, Florida, May, 2002.