Quality Assurance Recommendations for Open Source Developers

Bjarne Bue Jensen
bjarne@cs.aau.dk

Simon Lyngshede
simonl@cs.aau.dk

David Søndergaard
david@cs.aau.dk

Aalborg University
Fredrik Bajers Vej 7
DK-9220 Aalborg East

Abstract

This paper proposes a set of quality assurance methods which apply to different categories of open source projects. It includes suggestions to how these quality assurance methods are best implemented in an open source context. The paper presents the proposed set of methods as a model allowing open source developers to identify those methods most likely to be relevant in their given situation, based on the size and level of process control of their project. The result is a better understanding of how traditional quality assurance methods are equally relevant for open source development.

1 Introduction

Open source software has gained increasing attention from governments and organisations all over the world in the recent years. Several are considering a switch from their current solutions to open source alternatives. Such considerations raise a question: Is it possible for software written by volunteer developers under somewhat uncontrolled circumstances to be of the same quality–or even better–as its proprietary counterparts?

The answer is yes. There is a tendency to view open source software as a new and different approach to software engineering. In reality it is just another development method which must produce the same results as any other method to gain success. Proper quality assurance methods must be applied in order to achieve such quality. Open source developers, in particular those involved with large open source projects, are beginning to comprehend the necessity of such methods, but many smaller projects are still trying to get along without. Quality assurance requires planning and responsible management, which not all open source developers are ready to accept.

Several people have contributed to the literature on quality and open source software. In a recent paper a definition of quality of open source software was devised, which moved focus from the process to the product [11]. Jacqueline Stark has written a paper on peer reviews as quality management technique in open source projects, in which she concludes that peer reviews are more widely accepted by open source developers, than by traditional software developers, and that open source developers are committed to quality in their projects [19]. Eric S. Raymond is one of the most quoted authors in the field, he has written a number of essays on open source development, including "The Cathedral and the Bazaar" [16], in which he describes the open source development process and explains the mentality of open source developers. Feller and Fitzgerald have written a book on open source software, which also covers quality aspects. [9].

This paper introduces a set of quality assurance methods, known from traditional software engineering, and explains how they can be applied to open source development. Open source projects are categorised and recommendations towards appropriate quality assurance methods for each software category are made.

The paper begins with a brief description of the open source development cycle in Section 2. Section 3 presents the quality assurance methods and their usage in open source development and Section 4 categorises the software, and recommends when to apply which quality assurance methods. Section 5 concludes the paper.

2 Open Source Development

Generally speaking, traditional software development consists of a set of stages, which include planning, analysis, design and implementation. The open source development cycle is rather different from this. The planning, analysis and design phases are typically performed by the project founder (manager), and are not a part of the actual development cycle. It is vital for the survival of the project, that the project manager decides on a design solution before inviting additional developers to contribute. The design should be suited for distributed development, i.e. it should have a well-thought modular structure [9].

As a project matures, developers with various backgrounds and skills will begin contributing code. In the early

days of a project, it is up to the project manager to decide whether to accept or reject contributed code, thus he has the role of *gatekeeper* of the project. Developers who contribute large amounts of code may become *core developers*, or perhaps gatekeepers for specific areas or modules of the project.

According to Feller and Fitzgerald [9], the open source development cycle consists of the following phases:
Code ⇒ Review ⇒ Pre-commit test ⇒ Development release ⇒ Parallel debugging ⇒ Production release.

Every cycle starts with a developer writing code, which is submitted to the project for review. If the code is found worthy by the gatekeeper, it is tested to avoid breaking anything in the existing code. There are often no requirements to how test scenarios should be planned or performed, however, the gatekeeper's commitment to the project usually results in very thorough testing. When the code is tested and it is certified that it does not contain faults, it is incorporated in the development release, which typically means committing it to a version control system. This is when the actual debugging commences, as beta testers now can obtain the program and run it in their own environment. One of the strengths of open source development is its large number of beta testers, which result in early detection of errors, and fast bug-fixes [16]. After being tested for a while, the code is eventually merged into the stable production branch of the project, thus ending the life cycle of open source development.

3 Quality Assurance Methods

This section presents a number of quality assurance methods and explains how they are used in open source development, and how their usage could be improved. The methods are variations of those presented in the open source development cycle, as described in the previous section. We present the concept of change management and release management in open source development and explain how these can assist in improving the overall quality. We will compare pre-commit tests to unit testing and describe different approaches for unit test scenarios in open source development. Peer reviews and beta testing (parallel debugging) are described from an open source point of view as well.

3.1 Change Management

Traditionally the task of a change manager is to plan, monitor and confirm changes to the system, including keeping track of former versions of the software and which changes where made to which versions. The change manager must ensure that costs and benefits of change are properly analysed and that changes to a system are made in a controlled way [15]. An important part of change management is the *Change Control Board* (CCB). It consists of one or more people, for instance the project manager and representatives from other interest groups, e.g. software, hardware, engineering, support and marketing. The role of the CCB is to take a global view of the impact of the changes on the software and to consider the impact it will have on hardware, performance, customer's perception of the product, the quality of the product or reliability. Another of the CCB's tasks is to assign changes to developers and verify the changes, once they have been made.

The main problem with change management is that it must be balanced. Too much control will limit peoples creativity, as some changes may not be approved, while too little control can result in unreliable code [15].

3.1.1 Open Source Change Management

There are two types of changes in open source software, assigned and unassigned. Although the changes themselves are not much different, they must be described seperatly. Unassigned changes are not planned and are unknown to the project until they are submitted. The term *change* is used to describe both error corrections and functionality additions. Assigned changes are often in the form of bug reports or feature requests, which are accepted by a developer, who then implements the change.

Unassigned changes to open source software are made by individuals and not reviewed until they are submitted to the project. Detailed planning is typically limited in open source development, compared to traditional change management. This is possible because of the large amount of potential developers, it has both advantages and disadvantages: A good idea will not be abandoned because it is too expensive or impractical to implement, or because a manager misunderstands its purpose. On the other hand, many hours are potentially wasted, writing code that is not accepted, either because a change does not comply with the design of the project, or perhaps because somebody else has already made a similar change.

It is the nature of open source development to give people the freedom to explore any idea, they may have. Because there is less economical considerations to worry about, a project can afford to waste these developer hours, as long as there are people willing to spend them, however, steps should be taken to minimise the problem. A possible solution could be to use a bug tracking system to monitor and control new features, where users can submit ideas and developers can assign tasks to themselves. Another solution would be to require that suggestions must be submitted as test cases, in order to describe them properly. This approach is discussed further in Section 3.3.1.

Concurrent Versions System (CVS) is used extensively in open source development, to keep track of changes to the different components (files) of the system. It is important to limit the group of people with write access to the files in CVS, called maintainers. If the access group is too large, it will be difficult to control. On the other hand, if the access group is too small, the workload on the individual maintainer becomes too large. It is highly recommended to

divide the project into modules, which can be maintained by separate people.

When changes are submitted, the gatekeepers must verify that the changes to the software corrects the problem that was intended and that it is well written and documented. Patches should be rejected if the code is incorrect, poorly written or not accompanied by useful comments and documentation.

By applying change management principles to an open source project, developers gain a more structured approach of working with the source code. A bug-tracking tool can help organising and administrating the distribution of development tasks.

3.2 Peer Reviews

The term *peer review* is not clearly defined in the literature, it does, however, have a clear purpose: To find errors and defects in design or code, in order to correct these and improve the overall quality. Most people have difficulties detecting their own errors, a fresh pair of eyes is more likely to spot problems or come up with new ideas. A peer review might as well be an informal discussion about a technical problem, among co-workers, as a formal meeting with customers and managers. It is well known that the earlier in the development process defects are detected, the lower is its cost impact. Most literature agrees that a *formal technical review* (also knows as an *inspection*) is the most efficient way to detect errors before software is released [15].

A formal technical review needs to follow certain rules in order to be efficient. According to Pressman [15] a review meeting progresses as follows:

Three to five people should attend the meeting, namely the original author of the code under review, a review leader and two to three reviewers. All participants are expected to be well prepared. A review meeting should take less than two hours, so obviously only smaller segments of the software can be reviewed at a time. This has the advantage of increased focus on specific parts, and thus higher probability of detecting errors. The meeting begins with the author introducing his work, followed by a line-for-line walk-through of the code, during which the reviewers point out their comments and concerns. One of the reviewers act as recorder, and writes everything down for reference. At the end of the walk-through, the participants decide whether to accept or reject the code. If code is rejected due to errors, another review is required once it has been corrected.

Extreme Programming has taken peer reviews to a new level, by applying pair programming, which provides immediate review during development by a colleague. While this approach surely catches some errors that a single developer would not have caught, it is not as efficient as a formal technical review. The colleague may be "caught by the moment" and follow along on an erroneous design idea. XP compensates for this, by encouraging frequent refactorings of the code, thereby urging developers to re-think their ideas.

3.2.1 Peer Reviews in Open Source Development

Peer reviews are not an unknown phenomenon in open source development, but due to the distributed development method, formal technical review meetings, known from traditional software engineering, are virtually impossible. Instead, less formal reviews are practised, often, but not necessarily, by the gatekeeper of a given project. When a developer submits a piece of code, a gatekeeper must decide whether to accept it or not. This process is in fact a peer review, the gatekeeper looks over the code and determines if it is of a high enough standard, and fulfils the requirements, before determining to accept or reject it.

Should an error slip pass the gatekeeper, co-developers and beta-testers are ready to take over to perform their own peer reviews of the code. Raymond writes:

> *Given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix obvious to someone. Or, less formally, "Given enough eyeballs, all bugs are shallow."* [16]

In effect, this means that developers constantly review each others code, and whenever errors are introduced, they are quickly spotted and fixed.

Stark has conducted a survey from which she concludes that peer review is widely accepted among open source developers, and is in fact the most employed quality assurance technique in open source development [19].

The OpenBSD project is a good example of an open source development team, who has adopted peer reviews. Every piece of code is looked over by a co-developer, before being accepted in the CVS repository. If the reviewer does not find any problems with the code, he adds an "ok"-mark to the log file, and signs it with his name, thereby indicating that he has accepted it, and is equally responsible for errors in the code, as the author. Upon asked what the exact implications of the "ok"-mark was, an OpenBSD developer replied:

> *In theory, it is supposed to mean "I reviewed the change, understood it, tested it, and approve of the idea". In practice, it can mean some but not all of the above. Details vary with the nature of the change, what code it touches, and who "owns" that part of the [source code] tree.* [4]

Not all open source projects have as clearly specified a review policy, as that of OpenBSD. Smaller projects in particular are more loosely managed, and often relies solely on the project manager (who also has the role of gatekeeper) to perform reviews of submitted code. The implication of this is that code written by the manager himself is not reviewed. Even larger projects may face similar problems. Larger projects are often modularised, with different developers in charge of different parts of the code. Some

parts might be highly specific, and only have a single or a few developers working on it. Without a review policy, code which is never reviewed by other developers, may be added to these parts. Feller and Fitzgerald [9] point out a paradox that often occurs, namely that the simpler the code is, the more feedback will it get from a review, even though complex code would benefit more from the feedback.

A more managed approach to peer review is required, to avoid situations like those described. The OpenBSD project is, of course, an excellent example of working open source peer review, but perhaps the process can be improved further. Perpich *et. al* [13] explains how formal technical reviews can be performed in a distributed asynchronous manner without requiring a review meeting, by use of a web administration tool, and argues that such inspections are at least as effective as regular inspection meetings. Such a tool for reviewing could show useful in open source development.

Peer reviews can, if used properly, have an enormous impact on the quality of a program. During an efficient peer review, errors that would otherwise have made the software behave unpredictable are found before the software is released to users. Obviously, the less errors that exist in the released software, the better impression does the user get.

3.3 Unit Testing

In many traditional development methods, like the waterfall model, testing is often among the last events to be performed. More recent concepts, such as test-driven design and Extreme Programming, suggest testing to be an integrated part of the development process. These methods allow us to expose errors at an early point during development, and to deal with them accordingly. This is cheaper compared to testing and correcting errors after completed implementation [7, 18]. We apply a view on unit testing similar to that of Extreme Programming (XP) [7], in short, focusing on verification of the correctness and functionality of the smallest elements of a program.

Unit testing can be performed at different stages during development and by people with different roles. The developers will typically be encouraged to write test cases to help them debug. In XP, developers are required to write test cases before commencing programming of the actual code. The idea is that a test case will help them gain an understanding of the problem, while giving them a goal, in a sense a form of test-driven design. The goal is fulfilled when the test case runs without errors. Besides test written by developers, there may also be tests written by a quality assurance team or, in the case of XP, also by customers.

Controlled and structured unit testing is a valuable tool in regression testing, as applied in XP for example. After adding new code, all unit tests are run to ensure that the addition did not cause a defect in the existing code. By controlled and structured unit testing, we imply that there exists a process which guarantees that written unit tests are aggregated into persistent test suites, but also that the necessary test cases are written at all.

3.3.1 Unit Testing in Open Source Development

Observing open source projects indicates that only few have a policy on how to perform unit tests. Larger projects such as Mozilla [12] and PHP [10], which both have dedicated quality assurance teams, have testing strategies similar to unit testing. In the Mozilla project, test cases must be executed manually, and are for the most part designed to be run by end-users. Some test cases test for usability and stability across platforms, while others test the layout functions and how the rendering engine handles different Internet standards. The testing of PHP is automated and requires minimal intervention from users. Like with Mozilla, the test cases can be run by end-users, thereby running the same tests in a number of different environments. PHP has integrated the testing framework into its build tools, asking the users to run the tests after a successful compilation. Only in the case of a failing test is the user required to intervene and send feedback to the PHP quality assurance team.

Smaller projects rarely apply structured unit testing. The code is, obviously, tested by the author during development, and again by the gatekeeper upon submission, before it is merged into the project. Smaller projects could, however, benefit from a more structured approach to unit testing. We highlight three ways of introducing unit testing into an open source project, each with their own advantages and disadvantages.

Unit Testing with Heavy Communication

The implementation represented in Figure 1 is inspired by XP and demands that test cases are written before coding commences. A gatekeeper will need to review the test cases to ensure that they provide enough coverage. The gatekeeper can give the developer feedback on how to improve the tests, or he can accept the tests as is. When the tests are accepted, development can begin. As each unit is completed, the developer run test cases to verify that the code behaves as expected. If the design changes during implementation, the test cases must reflect this change. When the implementation is completed, the gatekeeper should verify that all supplied test cases run, and review the code, before adding it to the project.

As Figure 1 shows, there is a lot of communication involved in the process, which poses a problem in open source context, as the developers may be scattered around the world, meaning that a developer will have to wait for feedback from the gatekeeper. The situation does not improve if there are few gatekeepers and many developers, in which case the gatekeepers will spend most of their time reviewing and commenting on other peoples code. The main advantage of this model is that the gatekeeper can influence the design of the software, even before implementation begins, providing feedback on the test cases. Less experienced developers will benefit from the feedback from

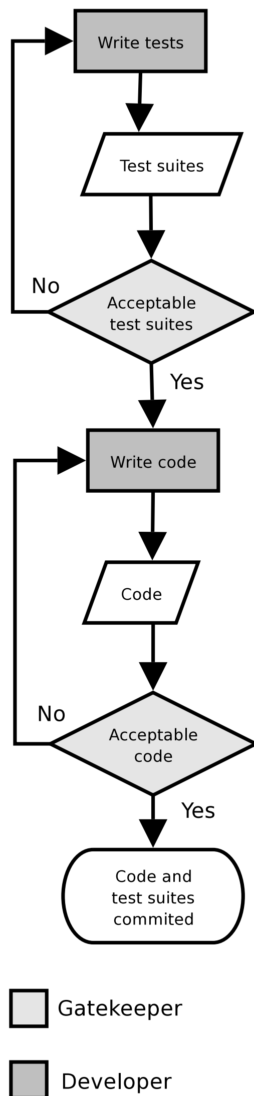

Figure 1: Unit testing with heavy communication.

more seasoned gatekeepers, leaving them with a better design before beginning implementation. The level of control introduced by this implementation is somewhat higher than some open source developers find to be comfortable, this introduces a risk that they may choose to leave the project. If the project can handle the extra communication, the result will be high quality code, even from inexperienced developers and generally a more uniform design. In practise, this will only apply to few projects.

Unit Testing with Reduced Communication
One way of reducing the communication between developers and the gatekeeper is illustrated in Figure 2. Here, the developer submits code along with appropriate test cases for the gatekeeper to review. If code is submitted without a proper test case, the gatekeeper should discard it. This model gives skilled developers more freedom to influence the design, but may restrain inexperienced developers from contributing larger modifications. Gatekeepers are not as burdened with reviews as in the previous model thereby getting more time for oher tasks. Some of this time, can then be used to help inexperienced developers getting their design right.

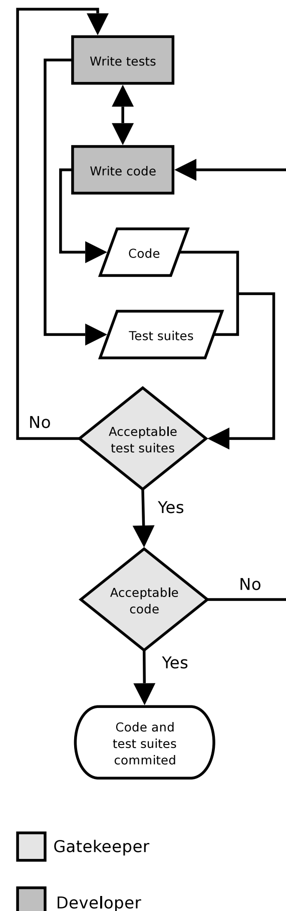

Figure 2: Unit testing with reduced communication.

Unit testing with Quality Assurance Team
The first two models assume scenarios where developers are willing to–and capable of–writing proper test cases. Open source developers can not always be expected to do this, and Figure 3 provides a solution. In this model, unit testing is performed by a dedicated quality assurance team (QA-team), developers and gatekeepers need only focusing on the actual program code. When a gatekeeper accepts a contribution from a developer, it is included in the project's development source tree, without additional testing. The QA-team writes test cases and reports problems to the developers. Untested code should not be included in a release.

The purpose of a QA-team is to produce test cases to illustrate errors in a program. Having a dedicated team performing this task takes responsibility away from the developers. It is difficult to identify own programming errors, an experienced QA-team will have better chances of detecting these. A QA-team would be able to produce better test suites, which in the end would uncover more errors. There

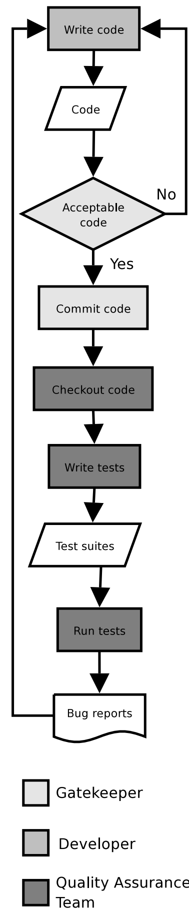

Figure 3: Employing a quality assurance team.

is little communication overhead in this model, the gatekeeper only need to accept or reject submitted code based on its functionality, and let the QA-team handle the testing aspect. Developers are relieved from thorough testing, which they may not have sufficient experience to do properly.

All this is naturally tied up on the assumption that the project is capable of attracting people with an interest in quality assurance. As shown in Figure 3 untested code may exist in the project. To avoid releasing untested code, the QA-team must coordinate with the people responsible for the eventual release to assure that everything has been properly tested before a release.

Unit testing has a profound influence on the quality of the software. The PHP project shows a prime example of how. By having users running test cases after compilation, the QA-team can verify that all parts of PHP functions as expected in a large variety of environments. A succesful compilation can even be seen as the first test case. Unit testing can reveal both deficient design solutions as well as implementation errors. Identifying these issues is the first step to eliminate them. Many quality problems are related to errors introduced by developers, therefore quality can be improved by reducing the number of errors within the software.

3.4 Alpha and Beta Testing

In traditional software development, a product will be tested by a team of professional testers in a closed environment, after development, in order to identify errors. This process is known as *alpha testing*. [14, 15].

Beta testing involves having end-users trying the software in a less controlled environment. Alpha testing is performed to avoid potential harm to the users equipment, but also because beta testing is not guaranteed to provide the same level or quality of feedback. Beta testing is often performed by handing out free copies of the software to trusted customers, or by selling it at a reduced price. The users can then report any bugs they may encounter, so the developers can deal with them, before shipping the final version [18]. The advantage of beta testing is that it is relatively inexpensive to get a large number of testers and obtain variations in the test environment. Beta testing is the only way of testing software on a large number of different platforms and realistic environments. Some users may try to use the software in a way the developers never imagined, and therefore did not take into account, which in turn will help to reveal problem areas in the code [18].

The problem with beta testing is that it can be hard to pinpoint exactly why software fails in one environment, and not another, because the developers do not have direct access to the beta tester or their equipment.

3.4.1 Beta Testing in Open Source

Alpha testing is rarely applied in open source projects. What little alpha testing there is, normally takes the form of an approval by a co-developer. An example of this is seen in OpenBSD, where another developer will need to approve new code, before it can be added to the project. This approval process functions as a sort of alpha testing in combination with review. The environment may not be as controlled as the one seen in traditional software development, but the testers are often highly skilled developers.

Beta testing in open source software has much in common with how developers get feedback in Extreme Programming. The concept of short release cycles is found in both development environments. Both Beck [7] and Raymond [16] describe how early and frequent releases to the end-users result in fast location of errors and suggestions of new features.

Naturally, open source projects have the same issues regarding feedback, as traditional software companies have. If the user chooses not to provide feedback, the beta testing is essentially worthless. Open source also suffers from an additional problem: Traditionally a group of professional testers will be performing alpha testing, open source projects can not rely on their testers to produce bug reports of a quality which can compare with that of the professional tester. What sets open source software aside from traditional software testing, is the type of feedback which can potentially occur. Because the source code is available

to the users, and because some of the users are developers (not necessarily on the same project) bug reports can be, and often are accompanied by a patch, which fixes the error [16]. This means that a patch can be developed and tested in the environment, where the error occurs, in turn allowing for fast and more precise fixes.

Large projects like Mozilla and Apache use bug tracking tools to manage feedback from their users. Bug tracking databases are a useful tool when distributing jobs among the developers. At the same time users can check that their problem have not already been reported and thereby eliminating duplicate bug reports. Small projects can do without a bug tracking tool, as there are less code and fewer developers, making internal communication with developers easier. The feedback provided by users can help, not only in the location of bugs, but may also provide ideas for feature enhancements.

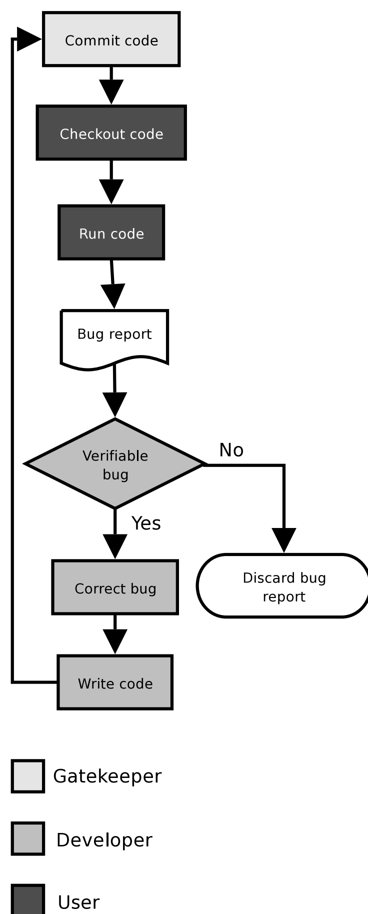

Figure 4: Open source approach to beta testing.

Figure 4 depicts the most commonly observed approach to beta testing in open source projects. A user obtains the software either directly from the source code repository or via a snapshot of the development code. While some users will download the beta version to assist in the testing others will want to run it to obtain new features. If the user obtains the source code, he will need to compile it, which is a test in it self, as pointed out in Section 3.3.1.

If the user encounters an error, he will write a bug report explaining the problem. A good bug report includes a description of the circumstances under which the error occurs and how to reproduce the error.

If the user provides a patch that remedies a bug, the developers will need to verify the bug and secondly inspect the patch. The user may think that a given feature is actually a bug or the bug may be a result of wrong use of the software. Patches from users must not be blindly accepted. An unreviewed patch could introduce new bugs or even a possible security risk.

3.5 Version and Release Management Definition

Version control and release management are the processes of identifying and keeping track of different versions and releases of a system. Version managers must ensure that the different versions of a system may be retrieved if needed and that older versions are not accidentally changed. A system release is a version of the system which is distributed to the customers. System release managers are responsible for deciding when the system can be released to the customers, for managing the release process and ensuring that the appropriate documentation have been written [18].

3.5.1 Open Source Release Management

Due to the short release cycles of open source software, which leads to many releases it is important to have a good version numbering scheme and a well defined strategy for releases.

Open source projects often have both a stable version and a developement version of its software. The developement version is changing frequently, so it will not be possibly to use release management to all developement releases. However, all major developement releases should be controlled by a release manager, the minor versions are often controlled by a CVS system. All stable releases should be controlled by a release manager, so changes can be verified and tested before a new release.

A release manager can either be static or dynamically chosen. When a release manager is static chosen, a single person or a small group have the responsibility for all the project's releases. This role would often be given to the project manager, or to a core group of developers. Static chosen release managers will build up experience over time, and become better at their job, but the project will come to depend on them, and will be impaired if they leave.

Dynamically release management is handled by a larger group, who take the responsibility for the release, when they feel the time is right. The group often consists of maintainers of the various modules. The advantage of dynamically choosing a release manager is that the project no longer relies on one, or a few persons to create the release.

A good release plan should lead to releases of higher quality, and a more predictable release schedule, which will benefit the users of the software. In addition, a consistent schedule will make it possible for the gatekeeper to better understand what his time commitment will be when he agrees to take the job of release manager [2].

There are two kinds of release plans often seen in open source projects. The first release plan, which we will call *fixed release dates*, means that there is a fixed date for when the next version is released. When using fixed release dates there should be a deadline for new features, a few months before the release date, after which there will only be made bugfixes and testing. New functionality will not be included in the release after this point. Several projects use such a release plan, including OpenBSD [3] and the GNU Compiler Collection (GCC) [2].

The other kind of release plan, which we call *fixed content*, means that there is a list of bugfixes, features and changes scheduled for the next release. Once all the desired features on this list is implemented and properly tested, a new version is released. Projects using this release plan include Mozilla [8] and the Apache HTTP server [1].

There are, of course, also projects with no release plan. They will just release the next version, when the developers feels they have made enough changes to the system to justify a new release. This requires a minimum of planning, however, it primarly works for small projects. Having a release plan can assure users that the software is under continous development, thus making them feel safer about using it.

Choosing version numbers from a well defined scheme, will make it easier for users to keep track of changes and updates without having much information about the projects. The version scheme should have a clear separation between numbers for the stable and developer version of the software, so it is clear for the user, which version they are running.

A well planned release strategy means that software will not be released to the public before it is tested, and the stable versions are reliable. From a public point of view, release management will guide them to reliable software and make sure that unreliable software is not released.

4 Quality Assurance Recommendations

This section presents a model that connects the quality assurance methods described in the previous section with different categories of open source software. The model is shown in Figure 5. Open source developers can use it to classify their development projects, and find recommendations of which methods to apply to their project.

The model presents four categories of open source software, *basic*, *extended*, *specialised* and *advanced*. Classification of a project into one of the categories is dependent on the project's size and level of control. Each category introduces a number of quality assurance methods, recommended for projects in that particular category–shown in bold font in the figure–and these methods propagates upwards and to the right. A plain font symbolises methods that are inherited from another category.

The recommended quality assurance methods primarily depends on the project's level of control. The higher control-level methods can not effectively be applied to projects with insufficient control. Also, some of the methods are only relevant for large projects. This is reflected in the model. Having a high level of control implies being willing to demand that developers follow a specific and potentially large set of rules and guidelines. The size of a project can be evaluated based on a combination of the size of the code base, the number of developers, the number of users and the complexity of the project.

It is not a goal for a project to be classified as belonging to the "advanced" category. Instead, projects should decide how large they could potentially become and how much control is needed to achieve their goals.

The following sections describe each category and which projects they apply to.

4.1 Basic Projects

The basic category recommends the use of

- lax unit testing,
- lax documentation,
- lax peer review,
- version control,
- and beta testing.

Projects in the basic category are small and exert little control on their development process. The low level of control means that only basic quality assurance methods can be successfully applied, however because of the small size, projects in this category have no need for more advanced quality assurance methods. For a large number of projects, the methods in this category are sufficient to ensure the quality of their software. Limited size and scope as well as the area of application are influential when deciding if the effort of applying stricter quality assurance methods can be justified.

Most open source projects use a version control system, in most cases CVS, without thinking of it as a tool for improving quality. Version control assist the developers by making the code more manageable, increasing maintainability. Implementing a version control system should be a key issue, as soon as there is more then one developer working on the code, and very often even single developers will be able to make good use of a version control system. The code in the version control system should be readable to anyone and writable to selected developers. Version control has the advantage of giving all developers constant access to the most recent code, which avoids that patches are made against older versions of the software. This avoids

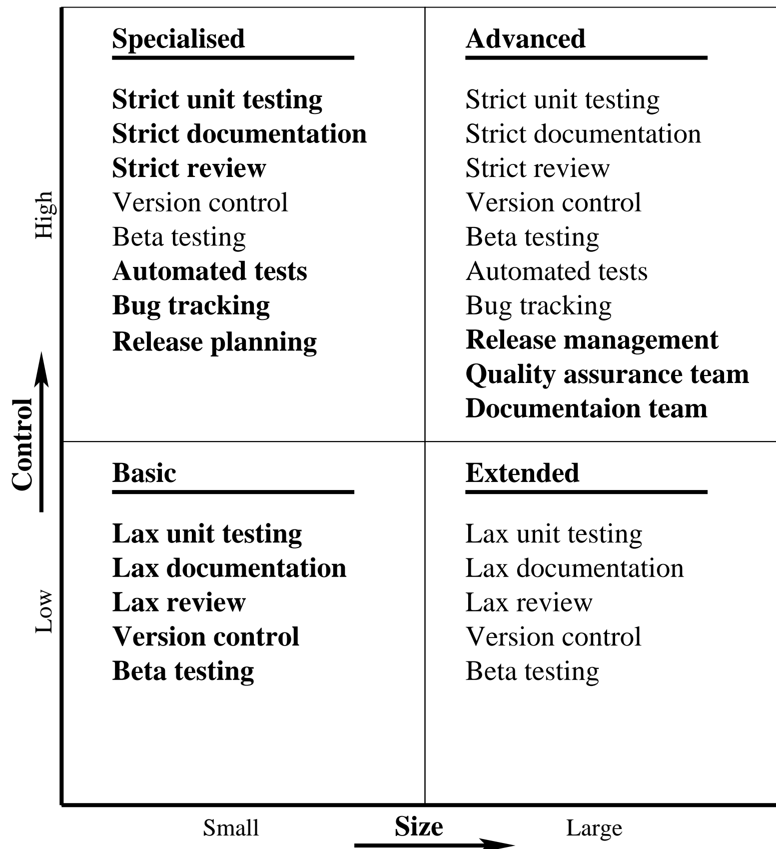

Figure 5: Quality assurance recommendations

extra work in getting the patches, to work with the latest version of the code [17].

Only trusted developers (maintainers) should be allowed to add new code directly to the project's code base, to avoid potential inclusion of malicious code. Patches from non-trusted developers are reviewed by maintainers, who will add the patch if it passes the review. A review is done by a single maintainer, who reads the code and attempt to verify that it does not contain any malicious elements, and to verify that it provides the claimed functionality. Code submitted by a project maintainer should be reviewed by another maintainer, if there are any, or by a trusted co-developer.

To ease the reviews and generally make the code easier to read and maintain, the project should apply coding standards, for instance the GNU Coding Standard [5]. Patches that do not follow the specified standard should be rejected. An advantage of selecting a popular coding standard is the availability of tools to assist developers in following the standard.

Testing is needed in every project, regardless of size or the level of control it has. As previously mentioned (Section 3.4), beta testing is widely used in open source development, it requires a low level of control, the software simply needs to be available for users to download. Users are requested to report any unexpected behaviour of the software.

Unit testing should also be applied to projects in this category.

One way of beginning unit testing is devising test cases for the central components first and then slowly start covering more and more parts of the software. Test cases should be bundled with the source code, and users who choose to compile their own versions should be encouraged to run the tests and report the results.

For most projects in the basic category, documentation is a secondary issue. There should, however, be at least some documentation, describing key aspects of the software, e.g. how to install and run it. Failure to provide such documentation may lead to potential users not being able to use the software at all, thereby loosing valuable beta testers.

4.2 Extended Projects

The extended category does not introduce additional quality assurance methods, as with the basic category it recommends the use of

- lax unit testing,
- lax documentation,
- lax peer review,

- version control,
- and beta testing.

Projects in this category typically started in the basic category and grew larger, i.e. attracted more users and developers. It is projects who do not wish, or need a higher level of developer control. The Linux kernel [6] is an example of a project in this category.

The quality assurance methods in the basic category have the advantages that they scale very well. The methods will often be even more effective, when having a large amount of developers and users. This is particularly true for beta testing, the more people using the software, the more likely is it that errors are encountered and reported. An increasing number of users results in a large amount of feedback. To deal with this, projects could define templates for providing feedback, and organise mailing lists for communicating with developers.

Version control systems have also shown to work very well with large projects. It enables a decent level of control with the source code, and allows many developers to work together, without interfering each other.

As for peer reviews, unit testing and documentation, it goes without saying that the more developers a project has, the more people will spend time reviewing, testing and documenting, thus improving the overall quality.

4.3 Specialised Projects

The specialised category recommends use of

- strict unit testing,
- strict documentation,
- strict peer review,
- version control,
- beta testing,
- automated tests,
- release planning,
- and bug tracking.

Projects in this category are often highly specialised software, for instance libraries, server applications or security related software. A common requirement of such projects is that the software must be very reliable. The development is highly controlled, in order to ensure that these expectations are met. An example of a project in this category is OpenSSH, an open source secure shell implementation.

In this category a large array of quality assurance methods are at disposal. Some of these are stricter versions of those recommended for use in the basic category, implying that more careful planning and performance is needed.

A peer review should be in the form of a complete walkthrough of the submitted code, and it should examine both the design and implementation. If the submitted code is a patch, the integration with existing code should also be reviewed. If possible, multiple developers should be involved in the review.

Unit testing should also become stricter in the specialised category. The recommendations in Section 3.3 suggest different approaches to how unit tests could be performed, for instance to require developers to submit test cases along with their patches–these test cases should be reviewed as well.

Automated testing can be applied in conjunction with unit testing, to ensure that no developers commit broken code, meaning code that violates interfaces or breaks existing functionality. Some open source projects uses so-called *nightly builds*, which is versions of the software that is built automatically by build scripts running every night. The build tools report compilation errors to the developers, who then need to inspect the relevant parts of the code in order to fix the problem. Automated testing can be any kind of testing which can be run without interaction. The key, and the reason for only introducing automated testing in projects with a high level of control, is that developers must respect the results of these automated tests and act accordingly.

Keeping track of bug reports, both from automated tests and users can be assisted by the use of bug tracking tools. These tools are designed to collect bug reports, help developers identify duplicate reports, keep track of the development process of a possible patch and be used as a tool for distributing tasks. Bug trackers are rarely needed in small projects as the amount of incoming reports is relatively low, this should however not stop small projects from deploying a bug tracker if they believe they can make use of it. The project should be aware that bug trackers can be difficult to use and administrate. The problems involved are related to assigning tasks, providing feedback and detecting inconsistency between the information in the bug tracker and the source code. Adding a bug tracker means adding extra bureaucracy to the project, which the developers must be able to administrate.

Small projects rarely have use for all the aspects of release management, however, especially with security and server application, a project should implement some release planning, to ensure the quality of the code released and to give users an idea of where the program is going. For many projects it is difficult to use deadlines, as the low number of developers, means that the project is more dependent on each developer, which may have other assignments. Therefore it would be advised to use fixed content releases, where the features required for the next release is agreed upon and worked towards, but without any promises as to when the next release will be out.

With a high level of control, projects should also be able to provide more in depth documentation, describing the different aspects of the software. The project must make it clear what level of documentation is expected to accompany a patch and reject any patch failing to deliver this.

4.4 Advanced Projects

The advanced category recommends use of

- strict unit testing,
- strict documentation,
- strict peer review,
- version control,
- beta testing,
- automated tests,
- bug tracking,
- release management,
- quality assurance team,
- and documentation team.

The category inherits quality assurance methods from the other categories and adds methods to improve organisation.

To belong in the advanced catagory, a project must be large and well controlled. Many of the largest open source projects are in this category, as large projects are difficult to manage without having a high level of control.

In very large projects it becomes impossible for one developer to keep track of all parts of the project. A way of limiting responsibility for individual developers is to divide the project into different teams, thereby creating smaller sub-projects with seperate management. This of course requires a highly modular design.

Besides development teams, specialised teams should be founded, to take care of documentation and quality assurance. A documentation team is responsible for documenting elements not directly related to the source code, often user manuals and guides.

The main purpose of the quality assurance team is to find ways to break the software! They should strive to produce test cases for every part of the code. Untested code should never be included in a production release.

Coordinating the work of multiple teams, who's effort eventually results in one combined product, requires careful planning, in order to ensure high quality of every individual part. Deadlines should be kept and milestones should be followed. A release manager should be appointed to coordinate the teams. He has the responsibility to indentify potential problems which could delay an upcoming release.

This concludes the recommendations for quality assurance methods applicable to open source software.

5 Conclusion

This paper has presented a set of practical suggestions to how established quality assurance methods can be applied to open source development. Each of these quality assurance methods require different levels of control of the development process, in order for a project to benefit from applying them. Some methods only become practical if the project is of a certain size as well.

A model was presented to identify which quality assurance methods are relevant for a project of a given size, and with a specified level of control. The model is a valuable tool for open source developers and project managers to assist them make qualified choices of which quality assurance methods to apply, thereby improving overall quality.

The paper shows that traditional quality assurance methods can be successfully applied to open source projects, with minor adjustments, and that open source projects can benefit from applying them.

Furthermore, it is suggested that some open source projects can achieve high quality by only applying basic quality assurance methods, and as such have no need for the more advanced methods.

It is our hope that this paper will direct the attention of open source developers towards the benefits of implementing well defined quality assurance methods and realise the importance of early quality management, thereby making quality the essence of the development process.

References

[1] Apache Release guidelines. http://httpd.apache.org/dev/release.html. Accessed May 12th, 2004.

[2] GCC Development Plan. http://gcc.gnu.org/develop.html. Accessed May 12th, 2004.

[3] OpenBSD FAQ. http://www.openbsd.org/faq/faq5.html. Accessed May 12th, 2004.

[4] OpenBSD Mailing List Reply. http://www.monkey.org/openbsd/archive/misc/-0403/msg01541.html. Accessed May 16th, 2004.

[5] The GNU Coding Standards. http://www.gnu.org/prep/standards.html. Accessed May 17th, 2004.

[6] The Linux Kernel. http://www.kernel.org/. Accessed May 30th, 2004.

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

[8] B. Eich and D. Hyatt. Mozilla Development Roadmap. http://www.mozilla.org/roadmap.html. Accessed May 12th, 2004.

[9] J. Feller and B. Fitzgerald. *Understanding Open Source Software Development*. Addison-Wesley, 2002.

[10] P. Group. PHP Quality Assurance Team. http://qa.php.net. Accessed May 11th, 2004.

[11] B. B. Jensen, S. Lyngshede, and D. Søndergaard. A Quality Definition for Open Source Software. In B. Wong, editor, *The Second Workshop on Software Quality, ISBN 0-86341-428-1*, pages 30–36, Edinburgh, Scotland, May 2004. The Institution of Electrical Engineers. Part of the 26th international conference on software engineering.

[12] T. M. Organization. Mozilla Quality Assurance. http://www.mozilla.org/quality. Accessed May 11th, 2004.

[13] J. M. Perpich, D. E. Perry, A. A. Porter, L. G. Votta, and M. M. Wade. Anywhere, Anytime Code Inspections: Using the Web to Remove Inspection Bottlenecks in Large-Scale Software Development. *International Conference on Software Engineering, Boston, USA*, 1997.

[14] S. L. Pfleeger. *Software Engineering - Theory and Practice*. Prentice Hall, 2nd edition, 2001.

[15] R. S. Pressman and D. Ince. *Software Engineering - A Practitioner's Approach*. Addison Wesley, european adaptation, 6th edition, 2000.

[16] E. S. Raymond. *The Catehedral and the Bazaar*. O'Reilly & Associates, Inc., revised edition, 2001.

[17] E. S. Raymond. Software Release Practice HOWTO. 2002.

[18] I. Sommerville. *Software Engineering*. Addison Wesley, 6th edition, 2001.

[19] J. Stark. Peer Review as a Quality Management Technique in Open-Source Software Development Projects. 2002.