Aspects of Software Quality of Open Source Projects

Päivi Raulamo-Jurvanen, Loinaz Merino Cerrajería & Pablo Alapont Tomás

Abstract

Success of open source software is claimed to be highly dependent on the quality of software (on top of the need for the software), and software quality, either as perceived quality or as an objective evaluation, can be an issue to consider when choosing among similar open source solutions. The idea was to study what is required from the open source software project itself and from enthusiasts contributing to it to ensure software of good quality and promote success of the project. The research method for this study was literature review. We reviewed related literature and all pointed to the fact that no method works if it is not used correctly, open source method being no exception. It seems that factors related to software quality in open source projects are related to people, processes and tools. People need to be predictive and do things right, preferably the first time. That requires thinking out of the box, being smart about how to do tasks and communicating plans and decisions (in detail) early on. We used Linux kernel as an example of successful open source project, considered being of good quality, for reflecting different factors. One important, but less emphasized fact was that open source model provides developers the possibility to live their passion, to have fun and to be happy, allowing them to
actually do their best. Another important factor affecting software quality in open source projects is the acceptance that outsiders may know better, and that better may be the true needs of the customer.

1. Introduction

In 1996, Eric Raymond wrote about surprising theories in software engineering which had emerged from the Linux experience (Raymond, 2000). Raymond was amazed how a project like world-class operating system Linux was run in such an awkward “bazaar style”, embracing a philosophy like “release early and often”, allowing part-time hacking and basically taking submissions from almost anyone. He concluded that a software program can be poorly documented, buggy and incomplete, but to succeed, the thing a project really needs is a plausible promise, a foreseeable future to attract the co-developers. Basically, the same observation was made by Stamelos, Angelis, Oikonomou, & Bleris (2002) who discovered that the industrial quality standards may not be met by a piece of open source software, but it can still provide further improvement opportunities. Popularity of open source software development, increased interest in and utilization of such software over the years have proven that good (and useful) software can be written in such “bazaar style”. How about the quality?

In traditional, closed-source projects the companies may try to ensure speed and quality of software development by employing only skilled personnel with suitable education and preferably years of experience. Furthermore, traditionally software development has been considered to be demanding headwork, requiring rigorous discipline and various processes. The term “bazaar style” describes the open source software development quite well. The background of a contributor may be unknown, in fact, it seems there is no discrimination based on the seniority of a developer. The things that count seem to be enthusiasm and motivation, especially the level of contribution rather than reputation or past expertise. Bosu and Carver (2013) reported that there is a high level of trust, reliability, perception of professionalism and friendship between peers of open source software having been collaborating in reviews for some time. Open source enthusiasts even claim that those being passionate about software development do their work the best and are expected to be highly productive when they produce software mainly for their personal satisfaction and especially - when they are having fun (Torvalds & Diamond, 2001; Stamelos et al., 2002).

It is clear that open source software development is far different from traditional approach. How does that affect quality of software - in comparison to closed-source (proprietary) software quality? How is quality of open source software perceived in general? Having fun while developing software is certainly not enough to ensure quality of software and success of the project. The idea is to study what is required from the open source software project itself and from enthusiasts contributing to it to ensure software of good quality and promote success of the project. Thus, the research question for the study is:
What are the key factors for developing open source software of good quality?

The research method of the study is a literature review. The main contribution of the article is to collect and compile knowledge and experience from literature for identifying perceptions of quality of open source software (in comparison to closed-source, proprietary software) and key factors for producing open source software of good quality. Such information may be useful to understand, especially when working in a (globally) distributed development team.

First we will discuss software quality in open source projects in general. Chapter 2.1 focuses on some existing software quality models and evaluation models as well as suitability of those to open source software development. In Chapter 2.2 we will describe processes and tools used (or preferred to be used) for ensuring software quality in an open source project. Chapter 2.3 presents other important factors affecting quality of software of open source projects. In Chapter 2.4., we will present a case of Linux kernel as an example of a successful open source project, considered being of good quality. In Chapter 3 we will elaborate the issue and discuss factors related to ensuring good software quality in open source projects. Chapter 4 discusses the limitations of the work briefly. Finally, in Chapter 5 we will conclude the findings and summarize the answer to our research question.

2. Software Quality in Open Source Projects

Open source software has been claimed to be one popular solution to “software crisis” by addressing costs, schedule and quality (Fitzgerald, 2004). Reasons for selecting and using open source software may vary from e.g. economic or security reasons to pure functionality related issues. Plausible promises or promising improvement opportunities may not necessarily be enough for an open source project to succeed. The findings from Lee, Kim and Gupta (2009) indicate that software quality and community service quality impact user satisfaction, and on the other hand, software quality and user satisfaction, in turn, impact the usage of open source software in general. Thus, they claim that success of open source software is highly dependent on the quality of software (on top of the need for the software, of course). (Lee et al., 2009.) It is claimed that quality, either as perceived quality or as an objective evaluation, can be an issue to consider when choosing among similar open source solutions (Adewumi, Misra, & Omoregbe, 2013; del Bianco, Lavazza, Morasca, Taibi, & Tosi, 2010).

It is good to keep in mind that we are all human, people make mistakes. The Heartbleed vulnerability (Codenomicon, 2014), a programming mistake in OpenSSL (used for securing traffic flow between servers and computers) which was discovered in 2014, has been claimed to be one of the biggest failures in the history of open source software development. Initially, a developer had forgotten to validate a variable containing a length in a new feature. Furthermore, the person reviewing the change (one of the core developers) failed to spot the defect and the erroneous software was eventually included into the release in the spring of 2012. Open source methodology or principle did not work in this case. It was a lesson to learn: for ensuring good
quality, open source software development should not rely on faith but only on verification and validation. (Vaughan-Nichols, 2014b.) After the Heartbleed incident, a company called Coverity (specializing in software quality and security testing solution) has checked open source software in large scale and claims that open source software has fewer defects than proprietary software. The findings from the report from 2013 claim software quality of open source C/C++ projects to be far better than quality of such proprietary software and Linux to be the benchmark for open source quality. (Vaughan-Nichols, 2014a; Synopsys, 2014; Synopsys, 2015.)

2.1. Quality Models & Evaluation Models for Software Quality

This chapter provides a short introduction to software quality models which may be utilized in evaluating quality of open source software. Software quality, product quality and process quality are important aspects of software development. There are a number of existing software quality models and standards that define or provide mechanisms to evaluate software quality. Those models provide decompositions of quality into more manageable sub-qualities. The term “software quality” is defined as capability of or degree to which the software satisfies customer or user needs, expectations or requirements (“ISO/IEC 25000:2014”, n.d.; “ISO/IEC 25010:2011”, n.d.; “IEEE 730:2014”, n.d.). Miguel, Mauricio, & Rodriguez (2014) claim ISO/IEC 25010:2011 to be the most complete among the basic models. ISO/IEC 25010:2011 (which replaced ISO 9126 model in 2007) specifically states “The quality of a system is the degree to which the system satisfies the stated and implied needs of its various stakeholders, and thus provides value” (“ISO/IEC 25010:2011”, n.d.).

There are a number of general standards available for software development as well, covering the software lifecycle, e.g. SQA – Software quality assurance IEEE 730:2014 and V&V – System and Software Verification and Validation IEEE 1012:2012 (“IEEE 730:2014”, n.d.; “IEEE 1012:2012”, n.d.). The former consists of means to monitor the processes and methods which are used for ensuring quality (software quality assurance) and the latter is the process of ensuring the system meets its specifications and fulfills its intended purpose (software quality control). There may also be additional compliance requirements in areas of law regulated industries (e.g. for automotive or medical software).

However, the existing models are claimed to be very general and difficult to apply to specific cases. For example, models for open source software specifically emphasize the participation of community members. (Adewumi et al., 2013; Aversano and Tortorella, 2015; Chawla and Chhabra, 2015; Miguel et al., 2014.) An important aspect, related specifically to open source projects, is communication, which should be considered as one of the factors affecting the quality (Miguel et al, 2014). Different approaches indicate a need to extend and adapt the existing standards for different purposes (Wagner, 2013).

2.2 Processes & Tools Used for Ensuring Software Quality

This chapter focuses on quality management of open source projects, i.e. processes and tools
used by professionals for ensuring software quality. As noted by Spinellis et al. (2009), open source projects provide transparency by having software source code, issue-tracking databases, mailing lists, wikis and associated data in version control system available either for all or just for registered users. Projects have also documented processes expected to be followed by contributing developers or other stakeholders. Thus, open source projects allow their output and processes to be evaluated. (Spinellis et al., 2009.) Dabbish, Stuart, Tsay, & Herbsleb (2013) point out the notable benefits of such transparency of development environment and processes. They claim that transparency of a development environment provides visibility of popularity, proves liveness of the project and enables evaluation or comparison of projects or community support within. One of the persons they had interviewed for their research had stated that usefulness of a project can be predicted by the size of community involved. Such information about community interest can be obtained e.g. from watcher or fork counts of a project and can be considered as an indicator of quality or value of the project. (Dabbish et al., 2013.)

Aberdour (2007) compares a number of ways open source software deviates from traditional notions of quality assurance and control, see Table 1. However, the practices are suggestive and do not comply with every closed-source or open source project. Thus, the comparison presented in Table 1 is not generalizable as such. For example, the practices described in Table 1 are idealistic for closed-source project and each of those may not be applied to every project in such extent. A stakeholder not involved in software development process of a company cannot assess those practices. On the other hand, for example, the first three listed practices in Table 1 (namely “development methodology”, “documentation” and “testing and quality assurance methodology”) may be well and thoroughly described for an open source project but completely overlooked by some closed-source project. As observable from Table 1, it seems some of the widely-known and claimed best practices from closed-source software development are (claimed to be) ignored by open source software development and some best practices are retained but executed rather differently. Good examples of open source projects with defined processes and practices are for example Linux kernel (“Linux Kernel”, n.d.), JavaScriptCore (“Webkit”, n.d.) and Ubuntu (“Ubuntu”, n.d.).

The governance of software development, being either closed-source or open source software development, needs processes that provide desired level of quality assurance and support for decision making. As Ciolkowski, Laitenberger, & Biffl (2003) defined it, to gain competitive advantage, software development should be pushed towards engineering-level precision by applying quality-enhancing techniques. The fact that contributors of open source projects may not know one another in neither personal nor professional level is extremely interesting. Some open source projects have established a system where changes will be reviewed by (at least theoretically) a large community of interested individuals. That type of activity of quality assurance is one of the keystones of open source software development, and empowers a widely known theory, known as “Linus’s Law” which states that “all bugs are shallow given enough of eyeballs” (“Linux Kernel”, n.d.). The “Linus’s Law” originates from the idea of pointing out that any problem will actually be, sooner or later, transparent to somebody. Linus Torvalds
refined the statement to emphasize the fact that some person will find the problem and most likely, it will be another person who will understand the problem in detail, in order to be able to fix it. While fixing the problem is important, finding the problem is claimed to be the bigger challenge of the two tasks. (Raymond, 2000.)

Table 1. Quality management in open source and closed-source SW development (Aberdour, 2007)

<table>
<thead>
<tr>
<th>Closed-source</th>
<th>Open source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Well-defined development methodology</td>
<td>Development methodology often not defined or documented</td>
</tr>
<tr>
<td>Extensive project documentation</td>
<td>Little project documentation</td>
</tr>
<tr>
<td>Formal, structured testing and quality assurance methodology</td>
<td>Unstructured and informal testing and quality assurance methodology</td>
</tr>
<tr>
<td>Analysts define requirement</td>
<td>Programmers define requirement</td>
</tr>
<tr>
<td>Formal risk assessment process - monitored and managed throughout the project</td>
<td>No formal risk assessment process</td>
</tr>
<tr>
<td>Measurable goals used throughout the project</td>
<td>Few measurable goals</td>
</tr>
<tr>
<td>Defect discovery from black-box testing as early as possible</td>
<td>Defect discovery from black-box testing late in the process</td>
</tr>
<tr>
<td>Empirical evidence regarding quality used routinely to aid decision making</td>
<td>Empirical evidence regarding quality is not collected</td>
</tr>
<tr>
<td>Team members are assigned work</td>
<td>Team members choose work</td>
</tr>
<tr>
<td>Formal design phase is carried out and signed off before programming starts</td>
<td>Projects often go straight to programming</td>
</tr>
<tr>
<td>Much effort put into project planning and scheduling</td>
<td>Little project planning or scheduling</td>
</tr>
</tbody>
</table>

Release management is one important issue in software development, in general, since releasing provides the software finally to the hands of end users. Fogel (2016) claims that open source projects tend to use more time and trouble for releasing process than closed-source projects. There are two typical approaches for releasing: time-based and feature-based releasing. Both approaches embed the process of getting a release branch into a releasable state. However, with time-based releasing, new versions of the software are released at a regular rhythm, with content ready for releasing, while with feature-based releasing the full features must be ready for releasing at some specific point of time. For a project with a relatively short release cycle the release planning is claimed to be easier than for one that follows a long release cycle. Such publication of regular releases is actually a coordination mechanism providing not only discipline and familiarity (regarding the release process) but also a reference point. It is claimed that time-based releasing strategy contributes to the quality of the software in open source projects which have little control of the contributors by providing a controlled development and release process. (Michlmayr, 2007.)
One of the most important things for an open source project is the hosting, the place to store the source code and related material for enabling development collaboration. Along with an online, publicly-accessible version controlled code repository, a hosting site may provide features for e.g. code review, bug tracker, development wiki, mailing lists, forums and build system. Although those features seem to be related to software development in general, those are related to software quality as well. The importance of those features and services lies not only in day-to-day activities for contributing developers but also in what is means to the people monitoring the progress of the project. (Fogel, 2016.) Most importantly, those features help to manage various changes introduced to the source code over period of time and thus contribute to software quality e.g. by providing traceability and change history for the files, and by enabling parallel development (branching and merging) and visible collaboration. Any extra tools used for verification and validation (like static code analyzers, possible configuration options or debugging tools) will provide confidence to the quality of those changes and of the project software in general.

2.3 Other Factors Affecting Quality of Open Source Projects

A large, sustainable community is claimed to be an enabler for fast software development. The onion model is claimed to be the most common model of such sustainable community. The model consists of different successive layers where each group member has specific responsibilities. In the center of the onion there is the small group of core software developers. The successive layers from the core represent the contributing developers, bug reporters and actual user of the software. The roles and responsibilities ensure the system is tested thoroughly, e.g. several times by different people and possibly on several platforms. (Aberdour, 2007).

Foucault, Palyart, Blanc, Murphy and Falleri (2015) studied mobility of developers (in and out of a project and also inside a project) and evaluated possible related impacts of such mobility on software quality. The study claims that external turnover (mobility in and out of a project) affects quality of the modules in a negative way while internal turnover (mobility inside a project) has little effect. However, they observed that there is no need to control such turnover in an open source project to improve the success of a project. (Foucault et al., 2015.)

Modularity of the software reduces complexity, and thus, impacts maintenance and promotes contribution. For open source projects modularity of the software is a crucial issue. Modularity of the software enables developers to contribute to the project without having to understand the whole core system. Stamelos et al. (2002) assessed the impact of software module size to the delivered quality measured through user satisfaction. They discovered that (up to some extent) the average module size was negatively related to the external quality of the software. They also reported that such partial relationship between module size and user satisfaction could be used when selecting contents for a release. If there were several candidates available for a specific software module, most likely the version to be selected for the release would the one with smallest module size. (Stamelos et al., 2002.)
2.4 Case Linux kernel- How they ensure quality?

There are a few impressive open source software projects that have been proven successful over time, e.g. Linux, Ubuntu, Docker, Apache and Firefox, to mention just a few. The list is by no means comprehensive, it just highlights some of the widely known and widely used open source projects. Whether an open source project is the best or is the most successful is basically a matter of opinion and can e.g. be an issue of debate (Pingdom.com, 2009; Huger, 2015).

Linux kernel open source project was chosen to be an example of open source project of good quality. Linux is popular, widely used and attracts contributors worldwide. Linux provided a strong design from the very start by being derived from Unix. The governance of Linux kernel is based on a concept of “chain of trust” and the project relies on version control tool “git”. In case of Linux kernel, there are so called gatekeepers, maintainers for different subsystems who are responsible for reviewing and accepting submissions to the subsystems. These gatekeepers are trusted by Linus Torvalds himself, the benevolent dictator of Linux. The reviewers are trusted to conduct reviews for quality, considering both functionality and coding guidelines. Reviewing code is emphasized to be beneficial for Linux kernel developers and the process as a whole. The reviewing task is seen as a useful way to learn how to program and to make a significant contribution to the process. (“Linux Kernel”, n.d.)

The project has concise and clear instructions how to proceed when someone is willing to contribute. Contributors are expected to have good C programming skills, be familiar with the known guidelines and specific standard coding style of Linux kernel. Nevertheless, any change to the software must be based on a true need and must be justified. Above all, the project emphasizes the importance of communication - specifically, detailed discussions with the right people. Before placing fingers on the keyboard for starting to program and making changes, a person is required to have hold discussions with the community concerning e.g. the exact problem to be solved, users affected by the problem, use cases addressed by the solution and the existing system falling short addressing the problem in question. (“Linux Kernel”, n.d.). The approach emphasizes the importance of understanding a problem, not just communicating it - thus, a problem needs to be explained clearly and simply.

According to the Linux kernel philosophy, untested code is considered to be broken code. Once the changes have been made, the developer is expected to test the changes (on at least 4 or 5 people, preferably more). Although the instructions do not mandate to use any specific tools the importance of utilizing debugging tools, static analysis tools and specifically, automated tools for ensuring quality is emphasized. Contributors are instructed to consider utilization of code checking tools and to remove any compiler warnings, possible potential sources of errors. A tool called “Coccinelle” is mentioned as a useful tool for finding a variety of potential coding problems and proposing fixes for those problems. Furthermore, to enhance reliability of memory handling, one is instructed to utilize a fault injection framework provided by the kernel. For
ensuring portability, the code should be compiled for other architectures, as well (cross-compilation). In cases of internal API changes, the responsibility of a developer is clearly stated - you fix what you break - and thus, to avoid any surprises, it should be ensured that any untouched code should be caught by the compiler. Developers are instructed to make clear, simple and logical series of patches and to provide descriptive commit messages and subject lines for those. The patches should be delivered to all those interested in receiving such emails, to take advantage of the power of the “Linus’s Law”. (“Linux Kernel”, n.d.)

So, in short, it seems the Linux kernel project has some basic guidelines in place and people should be well aware of their responsibilities. However, the management style of the project is far from traditional. The key thing is claimed to be avoiding decisions, at least big ones which cannot be undone or fixed afterwards. Any decision should be made small by ensuring the possibility to undo it. (“Linux Kernel”, n.d.)

3. Discussion

The extent to which various existing quality models are used in industry is not known. There are a few academic studies comparing different quality models, but not so much evidence how those would have been utilized in practice. (Adewumi et al., 2013; Adewumi et al, 2015; Miguel et al, 2014.) Aberdour (2007) and Nagy, Yassin & Bhattacherjee (2010) suggest that in order to find out whether some piece of open source software is of high quality and robust, one should get familiar with evaluation models developed by third-party organizations, such as Atos Origin, CapGemini, Navica or the Business Readiness Rating. Such evaluation models take into account a variety of factors (with different weights for each factor) including e.g. availability of training, documentation, third party support, integrated software and other professional services, community size, community age and lines of source code when estimating the maturity of open source software (Nagy et al., 2010). Such approach supports the definition of quality outlined by ISO/IEC 25010:2011 (“ISO/IEC 25010:2011”, n.d.) as the system needs to provide value by satisfying the needs of stakeholders. Quality may not always imply a number of detected defects, it can also be a reflection of e.g. user satisfaction or software performance.

Aberdour (2007) claims that high quality open source software relies on quality management - having open source developers who commit to sustainable communities and fully understand code modularity, project management and test process management. So, it is all about people and people-focused approach, those volunteers contributing to the project. Mockus, Fielding and Herbsleb (2002) point out two scenarios affecting software quality if the project does not widely attract volunteers. If the community is small and unable to fix defects on time, there is a risk that the core developers may be overloaded with finding and fixing defects (thus failing to develop new functionality) or the level of quality of the software will not be acceptable. Thus, it is recommended to build functional infrastructure as well as adaptable and extensive processes for comprehensive quality control. It is far better to be proactive than reactive, “take control of the results of your work instead of just reacting to problems that you hit” (Wagner, 2013, p. 195).
Considering project management, Aberdour (2007) claims that the processes of peer review and people management (which have an impact on software quality) are significantly different in open source projects than in closed-source projects. Processes of peer reviews and inspections in general require coordination and communication, especially in distributed software development and when people are not known to one another. There is a concept known as “Conway’s law” which emphasizes the need for coordination in software development. The design of the software is claimed to resemble the communication structure of the organization, thus the organization should be structured according to the need of communication. (Conway, 1968.) The people responsible for the subsystems or parts of the software need to be identifiable and must actively monitor ongoing communication activity.

Transparency of a development environment of an open source project provides visibility of popularity and proves liveness of the project. Such transparency also enables evaluation or comparison of projects regarding e.g. usefulness or community support within. Such transparency is not provided by and such evaluation is not normally possible in cases of commercial companies. Visibility of popularity and liveness of a project can be considered as indicators of quality or value of the project. The development approach (e.g. agile) for a commercial company may be known but most likely tools, processes and methods are not revealed to outsiders. Thus, for open source projects the quality can be not only experienced by utilizing the software but also by reviewing or evaluating the actual source code or related processes. The concept of transparency of open source projects seems to be in line with the principles of agile software development which emphasize individuals and interactions over processes and tools, although some of the latter are vital too.

Open source software (as any software) may have a long life, like in the cases of Linux and Apache. Over time some pieces of such software may be forked by even thousands of developers. Lehman (1980) and Lehman, Ramil, Wernick, Perry, & Turski (1997) have formulated laws of software evolution which describe a balance between forces driving new development and slowing down development progress. Lehman (1980) defines operating systems (such as Linux) as E-type programs which are claimed to perform real-world activity and to be extremely change prone. The laws of software evolution (Lehman et al., 1997) include eight statements abstracting the observed behavior. The seventh law of software evolution is defined as “Declining Quality: The quality of E-type system will appear to be declining unless they are rigorously maintained and adapted to operational environment changes” (Lehman et al., 1997, p. 21). Surprisingly, the analysis indicated that there was no data providing evidence neither for nor against that specific statement about declining quality (Lehman et al., 1997.) Israeli and Feitelson (2010) studied 810 versions of the Linux kernel (from March 1994 to August 2008) to characterize the system’s evolution using the Lehman’s laws as a basis for their study. For Linux they discovered that the continued, preventive work had actually prevented the decline of quality and increased usefulness of the software to the users. However, they concluded a deeper study would have been required to verify the law of declining quality. (Israeli & Feitelson, 2010.)
Linux kernel has been proven to be a successful example of a large software system being developed using open source development model. Godfrey and Tu (2000) pointed out system growth and success of Linux while prior research suggested that growth of such large system would generally slow down (Lehman, 1980; Lehman et al., 1997). Modularity is considered as an important factor for code quality. According to Godfrey and Tu (2000), roughly half of the entire source tree for Linux kernel consists of relatively independent device drivers which are parallel features, specific to a particular CPU. Thus, not all Linux kernel source code is interdependent. They estimate that a compiled version of Linux kernel will include only roughly 15% to 50% of the software of the entire source tree. (Godfrey & Tu, 2000.) Another study claims that the overall code quality of Linux kernel has been improving over time, although new defects are constantly being introduced to the system. However, tools (under-exploitation of existing tools and need for tools which would integrate better to the development process) and reactivity of maintainers are seen as potential problems for improving code quality. (Palix et al., 2011.)

4. Limitations

The main limitations of the work are in the material and examples selected for the study. The literature review was not done in a systematic manner and does not include all relevant research on the topic. Also, we focused on analyzing the process of developing Linux kernel, not including other similar cases or other types of projects for comparison in more detail. However, since the topic was to analyze factors affecting software quality, Linux kernel was seen as a suitable option for this purpose. Linux kernel project has highly collaborative, distributed and enthusiastic set of contributing developers, is widely used and is considered to be of high quality, in both terms of user perception and code metrics.

5. Conclusions

The aim of the study was to answer the following research question: What are the key factors for developing open source software of good quality? Apart from project infrastructure and software modularity, it seems the factors are related to people, processes and tools. A large, sustainable community of dedicated and passionate stakeholders having specific roles and responsibilities and disciplined, quality-focused governance of software development form the foundation for software quality of open source projects. Defined, known processes need to provide for desired level of quality assurance, coordination and support for decision making. Guidelines and communication (with the right people) enable distribution of information and understanding of the problems and related solutions. Above all, people are expected to obey the rules and be worth trust.

However, no method works if it is not used correctly and open source method is not an exception. The slogan “quality is free”, many times trumpeted by enthusiastic agile developers,
does not imply to being free in a sense that you would gain quality by doing nothing or not caring about quality. What it actually does mean is thinking out of the box, being smart about how to do your tasks and communicating your decisions in detail early on - thus, being predictive and doing things right, preferably the first time. It seems like the quality aspect of the concept of “chain of trust” in open source projects is focusing on trusting proper quality assurance methods used by the developer community while the same concept in closed-source projects is focusing on (desired) quality built in to software by developers themselves.

All in all, open source method, the way of working resembles the concept of “wisdom of the crowds”. The concept embodies the idea of collective opinion that may be, under the right circumstances, smarter than the idea of an individual. There is a clear point that aggregation (not centralization) of information and judgement is needed, and the information needs to be shared. It is even claimed that disagreement and contest, rather than consensus or compromise, produce the best collective decisions. (Surowiecki, 2005.)

“Quality is never an accident. It is always the result of high intention, sincere effort, and intelligent execution; it represents the wise choice of many alternatives - choice, not chance, determines your destiny.” (William A. Foster)

References


Mockus, A., Fielding, R.T., & Herbsleb, J. (2002). Two case studies of open source software development: Apache and Mozilla. ACM Transactions on Software Engineering and


