Towards a Unified Definition of Open Source Quality

Claudia Ruiz and William Robinson

Georgia State University, Computer Information Systems Department,
35 Broad Street NW, Atlanta, GA 30303, USA
{cruiz5,wrobinson}@gsu.edu

Abstract. Software quality needs to be specified and evaluated in order to determine the success of a development project, but this is a challenge with Free/Libre Open Source Software (FLOSS) because of its permanently emergent state. This has not deterred the growth of the assumption that FLOSS is higher quality than traditionally developed software, despite of mixed research results. With this literature review, we found the reason for these mixed results is that that quality is being defined, measured, and evaluated differently. We report the most popular definitions, such as software structure measures, process measures, such as defect fixing, and maturity assessment models. The way researchers have built their samples has also contributed to the mixed results with different project properties being considered and ignored. Because FLOSS projects are evolving, their quality is too, and it must be measured using metrics that take into account its community's commitment to quality rather than just its software structure. Challenges exist in defining what constitutes a defect or bug, and the role of modularity in affecting FLOSS quality.

Keywords: open source, software, quality, measurement, literature review.

1 Introduction

Quality is extremely subjective, with as many definitions as there are people with opinions. It is no surprise that studies evaluating the quality of FLOSS and comparing it with traditionally developed software have produced mixed results [1], [2], [3], [4]. This is probably because of two main reasons: each study has defined quality differently and has evaluated it using different characteristics of different FLOSS projects. Defining quality differently will of course produce mixed results, but even when studies define quality in similar terms, they evaluate it using dissimilar projects and compare different project characteristics.

In order to understand what it is about certain FLOSS projects that lead them to produce high quality software, the antecedents of FLOSS quality must be found. There is, however, no current research on the antecedents of FLOSS quality [5].

This paper takes the first step towards addressing this issue by reviewing the FLOSS literature order to understand how is quality being conceptualized and to propose a unified definition of FLOSS quality.

The rest of the paper is organized as follows: sections two and three provide a brief background on FLOSS and software quality; section four presents the methodology followed; section five describes the findings, and section six discusses the implications of the findings., categorizing the findings according to research approach, and definition of quality.

2 FLOSS

FLOSS has grown dramatically in the 2000s and is an integral part of the IT industry. It directly supports 29% of the software that is developed in-house in the EU and 43% in the US and could reach a 32% share of all IT services by 2010 [6].

Linux, Apache, Firefox are commonly found in many computers today and were developed using open source models. Apache is a Web server used by 60% of Websites worldwide [7] and 23.2% of European and 14.5% of North American Web surfers use the Firefox Web browser [8].

This growing popularity begs the question: is FLOSS *better* than traditionally developed software? Traditionally developed software projects are considered successful if they finish on time, on budget and meet specifications. But the same standards cannot be applied to judge the success of FLOSS projects, since they usually have minimal budgets, are always in a state of development, do not have an official end time, and do not have formal specifications [9].

This lack of objective measures of success has not deterred the adoption of FLOSS products. It even has become a common assumption that FLOSS products are higher quality than traditionally developed software [10], [11] with firms entering FLOSS projects citing FLOSS's "quality and reliability" as one of the main motivating reasons for the endeavor [12].

This assumption can be traced back to Linus Torvalds, the architect of the Linux kernel, who said that "given enough eyeballs, all bugs are shallow" [13]. Torvalds believed that FLOSS's public peer review and frequent releases lead to fewer bugs because there are more people looking at the software, reporting errors, and fixing those errors. This assumption has a kernel of truth: it has been observed that most problem (bug) reports and solutions in FLOSS projects are contributed by periphery community members and not so many by the core developers [14].

A FLOSS project is one that offers its software product's license in accordance to the Open Source Definition [15] providing for free redistribution of the compiled software and the openly accessible source code.

A typical FLOSS project is composed by a community, whose structure has been described as being like an "onion" with the most actively contributing members, who are the most invested in the project and have the greatest decision power in the inner part and the least contributing members with the least amount of decision power on the outside. The project leader is at the center and radiating out are the core members, the active developers, the peripheral developers, the bug fixers, the bug reporters, the readers, and the passive users [16]. These roles are dynamic, changing as the community evolves as the system they are building evolves [16].

Although each FLOSS project is different and has different development practices and processes, The Apache project can be used as a model of a mature FLOSS development project given its success and well documented and researched development cycle stages. Its stages are below:

Identifying work to be done. Core developers look at the bug reporting database and the developer forums for change and enhancement requests. The core developers need to be persuaded of the priority of the request for it to be included in the agenda status list.

Assigning and performing development work. Core developers look for volunteers to perform the work. Priority is given to code owners (those who created or have been actively maintaining the particular module). The developer then identifies a solution and gets feedback from the rest of the developers.

Prerelease testing. Each developer performs unit testing of his/her own work. There is no integrated or systems testing.

Inspections. Each developer then commits his/her changes and the code is then reviewed before it is included in a stable release, while changes to development releases are reviewed after being included in the release.

Managing releases. A core team member volunteers to be the release manager and makes the decisions pertaining to the individual release. He or she delineates the scope of the release by making sure that all open requests and problems are resolved and restricts access to the code repository to avoid any more changes [17].

These development cycle stages draw a parallel to the Scrum Agile development methodology, where a product owner creates a backlog, a prioritized list of functional and non-functional requirements for building into the product. Development is performed in sprints which are 30 day iterations of development activities, which include only the highest priority backlog requirements that can be successfully completed in the allotted time [18].

Most FLOSS development practices are very similar to Scrum Agile development methods but less structured and more ad hoc.

3 Software Quality

The origins of software quality can be traced back to industrial engineering and operations management and their development of product quality concepts and quality management practices. For these fields, quality is adherence to process specification [19], [20] in order to produce a product that meets customer requirements with zero defects [21], [22]. In order to achieve this goal, approaches such as TQM (total quality management) [23], [24] were developed to integrate quality into all company activities and Six Sigma to measure for quality [25].

Industrial engineering and operations management's view of quality can be categorized as the manufacturing, user, and product approaches to quality as described by [26]. Table 1 summarizes these definitions of quality as well as others (transcendent and value). As well as categorizing definitions of quality, Garvin also categorized the eight dimensions of product quality (Table 2) [26].

Table 1. Garvin's quality definitions [26].

Approach	Definition of Quality
Transcendent	Innate excellence that cannot be defined, only recognized through experience.
Product	Discrete and measurable product characteristics.
User	Subjective consumer satisfaction.
Manufacturing	Conformance to specification.
Value	Conformance to specification at an acceptable cost or price.

With this legacy from industrial engineering and operations management, software quality started with the product definition of quality by defining frameworks of factors. The most popular were Boehm's model of 23 factors (dimensions of quality) [27] and McCall's with 11 factors [28], [29] which are all listed in Table 2.

Both of these frameworks of factors left out the measure (actual thing that is counted) for each factor. Each implementer and developer was left to define his or her own metrics and criteria for each factor. The ISO 9126 [30] Information technology – Software product evaluation: quality characteristics and guidelines for their use, which is part of the ISO 9000 set of standards by the ISO (The International Organization for Standardization) for quality management, was an attempt to standardize the quality factors to six main factors with three sub-factors under each one.

Table 2. Quality Factors

Model	Factors
McCall	Accessibility, Accountability, Accuracy, Augmentability, Communicativeness, Completeness, Conciseness, Consistency, Device-independence, Efficiency, Human engineering, Legibility, Maintainability, Modifiability, Portability, Reliability, Robustness, Self-containedness, Self-descriptiveness, Structuredness, Testability, Understandability, Usability
Boehm	Correctness, Efficiency, Flexibility, Integrity, Interoperability , Maintainability, Portability, Reliability, Reusability, Testability, Usability
ISO 9126	Efficiency, Functionality, Maintainability, Portability, Reliability, Usability
Garvin	Aesthetics, Conformance, Durability, Features, Perceived quality, Performance, Reliability, Serviceability

While the quality factor frameworks were used to assess the software product, process frameworks were developed to assess the quality of the process producing the software and to accommodate the manufacturing definition of quality [26]. One such framework is the CMMI (Capability Maturity Model Integration), which is a process improvement framework that can be used to drive organizational change and to judge the process maturity of another organization. CMMI has five levels, with one being the lowest. A level five organization is one where processes are defined (level 3), quantitatively managed (level 4), and are continually being optimized (level 5). This maturity model lists the processes that an organization should have in order to be considered at a certain CMMI level but it leaves the details of how to put them into place up to the organization.

Quality factor frameworks come close to Garvin's product definition of quality because they distill quality into a set of measurable characteristics while the process maturity models most closely resemble Garvin's manufacturing definition of quality because they define quality by evaluating how close an organization's processes meet a predetermined specification. Garvin's user definition of quality, on the other hand, is hard to implement for commercially offered software because a user's satisfaction or rather dissatisfaction with a software's features or performance cannot be immediately addressed. Rather, user satisfaction must be bundled with the problem resolutions and new feature requests of all other users into a new release, patch, or service pack, which are infrequently issued due to their cost. Because of this limitation, software quality has become Garvin's value definition of quality: conformance to specification at an acceptable cost.

In contrast, FLOSS software quality most closely fits Garvin's user definition of quality. Users can directly log problem reports and new functionality requests directly into the software project's issue tracking system that is used by developers. Because FLOSS has frequent releases, those requests can become part of the software much more quickly than commercially offered software, thus better satisfying users.

4 Methodology

In order to answer the research question, how is quality defined in the FLOSS literature, we performed a literature review. Since the definition of quality is very subjective, we adopted an interpretive approach [31] to this review by applying a grounded theory methodology [32].

We used the *Straussian* type of grounded theory in order to allow previous theories and our own interpretations of quality to guide the data collection and analysis [33].

4.1 Data Collection

In order to comply with the theoretical sampling necessary in grounded theory, we searched Google Scholar for journal articles and conference papers containing the terms "open source" and "quality". We retained papers that met the following criteria: explicit definition of quality and empirical validation of the quality definition. We decided these criteria would provide a relevant sample because the authors of these papers would have to explicitly define quality and operationalize it in order to empirically validate it.

This process left us with 24 papers, to which we then added 16 from the quality and defect-fixing categories in [34] that met the above stated criteria.

This left us with 40 papers that defined quality and performed some form of empirical validation of that definition.

4.2 Analysis

The papers of this literature review were analyzed using open, axial, and selective coding [32]. As the papers were read, they were coded using open coding Text segments from each of the papers were highlighted and labeled with a code to categorize and conceptualize the data.

The open coding phase produced 75 codes, which were used to label 637 text segments from the 40 papers gathered. The codes reflected how the authors defined quality, the measures used to operationalize it, the research methods used to analyze it, and the characteristics considered in the FLOSS projects that were used to validate their definition of quality.

The axial coding phase produced five codes which were categories containing the codes from the open coding phase. From the 75 codes from the open coding phase, four were discarded because they labeled few text segments and did not help explain how quality is interpreted in FLOSS research.

Table 3 show the categories produced from the axial coding phase. The categories were based on how the authors approached the research, how they analyzed the data, and how they defined and operationalized quality, with the two main categories being quality as a process and quality as a product. The final category deals with the type of data sampled by the authors to validate their models, in this case, the characteristics of the FLOSS projects they examined. These categories were chosen because they follow the research process: an approach must be chosen along with an analysis method; the phenomenon of interest must be operationalized and finally, the data sample must be chosen.

The final phase, the selective coding phase, produced and integrated category that narrated the conceptualization of quality by FLOSS researchers. This phase was complete when theoretical saturation was reached, meaning, not new conceptualizations could be obtained from the data.

Table 3. Categories from Axial Coding

Category	Description	Sample of codes within category
Research approach	Approach used to analyze quality in research study	Case study Survey Factor model Maturity model
Analysis method	Methodology used to analyze data	Regression Structural equation model Machine Learning Social Network Analysis
Quality as a process	Quality operationalized as processes that can be measured.	Defect fixing rate Defect fixing time Definition of bug Quality assurance procedures Process metrics
Quality as a product	Quality operationalized as characteristics of final software product.	Product metrics Number of post-release defects Cyclomatic complexity Halstead Volume CBO (coupling between objects)

Examined project	Characteristics considered of FLOSS projects used to validate operationalization of quality	Maturity Popularity Number of developers Development time examined Software type Version

5 Findings

In this section, we describe how researchers interpret quality in FLOSS publications.

5.1 Quality as a product

These studies defined quality as structural code quality [1], [35], [36], [37], [38], [39], [40]. Metrics that are used to measure structural quality are number of statements [1], [35], cyclomatic complexity [1], [35], [36], [38], [39], [41], [42], number of nesting levels [1], [35], [38], [43], Halstead volume [1], [35], [42], coupling [35], [36], [37], [41], [43], coding style [35], statements per function, files per directory, percentage of numeric constants in operands [35], growth of LOC (lines of code)[38], modularity [2], average coupling between objects, cohesion, number of children, depth of inheritance tree, methods inheritance factor and other internal software structure metrics [1], [35], [36], [37], [38], [39], [42].

The idea behind measuring software code structure is that well-designed software is less complex, less likely to contain faults, and easier to maintain [42].

Measuring code structure left the researchers with more questions than answers. The most successful projects in terms of number of downloads and popularity were not the ones with the highest structural quality [42], [43]. Another study found that the software modules with the highest rate of change were not the ones with the highest structural complexity [38]. Even using machine learning algorithms with structural quality measures in order to predict faults did not produce clear results [39].

Comparing structural quality between open and closed projects produced mixed results with some studies finding that FLOSS projects had quality comparable to closed projects [44] while others found that open source software did not prove to have structural code quality higher than commercial software [1], [35].

Using structural quality to define, measure, and compare FLOSS quality has not proven effective with different researchers achieving different results even when using the same metrics.

5.2 Quality as a process

5.2.1 Defect Fixing

Defect fixing [2], [3], [38], [39], [45], [46], [47], [48], [49], [50], [51], [52], [53], [54], [55], [56], [57], [58], [59], [60], [61] is by far the most popular definition of quality as a process in the FLOSS quality literature. Authors have done studies defining the what constitutes the process itself in order to determine how it works [38], [48], [49], [50], [51], [52], [53], [54], [55], [61] and developed models to test its effectiveness [45], [62].

They have approach it in terms of total bugs fixed [37], [48], [49], [58], [59], [61] and speed of bug resolution [3], [46], [47], [51], [61]. These approaches take into account the evolving nature of FLOSS, which is never truly finished, but rather, remains in a permanently emerging state. It also considers that FLOSS testing and defect reporting and fixing is a community activity where developers, users, and periphery members collaborate to create the software.

However, results using this approach have been mixed. Some open source projects resolved service requests more quickly than their closed counterparts, others did not [3]. For other studies, software type (database, financial, game, networking) made more of a difference in determining defect resolution speed along with number of developers (groups with less than 15 developers were the most efficient) [47]. The main difference with closed software is that in most FLOSS projects, bugs are only addressed after feedback is received from users. There is no way to measure the quality of a release pre hoc, only ad hoc [50]. However, this attitude is changing with projects such as GNOME, Debian, and KDE forming their own Quality Assurance teams and enforcing quality assurance tasks [57].

Even though the defect fixing approach to measure FLOSS quality considers the evolving nature of its quality, research using it has not operationalize it in an evolutionary manner. Most studies have looked at the bug databases of FLOSS projects cumulatively after a certain amount of time (i.e. after six months of activity) rather than looking at defect resolution rates per release (except for [39], [53], [58] which did compare product releases), which is the evolutionary cycle of FLOSS software. The studies that looked at defects per release found that FLOSS has lower post-feature test defects than commercial software but higher post-release defects than commercial software [58] and that release software quality is cyclical, with the Mozilla 1.2 showing a major decrease in quality, which was improved in later releases [39].

The cyclical nature of FLOSS quality is illustrated in a study that showed that bug arrival rates follow a bell curve through time between releases. Whenever drastic changes were introduced to the software, the rate would also drastically change [53]. This would suggest that defect rates and thus defect resolution rates vary across releases depending on the changes being introduced. If the release introduces new features, or makes major changes to the architecture of the software, many defects will be introduced, while those that simply introduce defect fixes and enhancements, will introduce less.

Another issue stems from the definition of bug. Bug reports in bug databases in FLOSS project management Web sites could include anything from “failures, faults, changes, new requirements, new functionalities, ideas, and tasks”[49]. Not to mention duplicate bug reports, poorly defined ones, and those that are out of scope with the product [56]. This happens because bug reporting systems are usually open to the public, and users without enough technical skills will make mistakes writing the bug reports [57].

Measuring defect-fixing effectiveness in FLOSS projects has provided mixed results because different studies have defined and thus measured defects or bugs differently. They have also calculated defect-fixing rates by looking at bug tracking databases in the cumulative, without considering that defects are introduced and fixed cyclically in FLOSS, per release.

5.2.2 Other Processes

FLOSS projects tend to rely on tools to enforce policies and standards [56]. Such tools include defect tracking systems, version control, mailing lists, automatic builds, etc. [55], [56], [57].

One quality assurance activity that is performed in traditional software development, peer review, is done differently in FLOSS projects with successful results. Peer reviews in open source were more efficient because there is no time wasted scheduling meeting since people work asynchronously and have more detailed discussions [14]. An example from the Apache project shows that it has three types of peer review procedures, depending on the experience and trustworthiness of the developer [14].

Despite the successful inclusion of peer reviews into the FLOSS development process, testing procedures have not managed to make the necessary cross over into FLOSS. Most projects do not have a baseline test suite to support testing, this means no regression testing can be performed [54].

Developers perform their unit testing and do sometimes better than commercial software [58], but it is up to the users to discover bugs and defects which could be eliminated with system or integrated testing.

5.2.3 Process Maturity Models

Maturity assessment models have been formulated to help users and integrators evaluate the quality of a FLOSS project versus another [59], [60], [63], [64], [65], [66], [67], [68]. These models provide a set of criteria to evaluate a FLOSS project. Different models concentrate on different criteria, but they all provide a way to quantify and evaluate the quality of a FLOSS product.

Maturity models use organizational trustworthiness as a proxy for product trustworthiness and thus quality—if the product is built correctly, it will then have high quality.

The assessments are mostly for the FLOSS integrator who must assess the risk of adding the FLOSS product to his or her existing architecture. The assessment models are not predictive (they do not evaluate the factors that lead to quality, nor do they provide a construct for quality) they simply provide a set of criteria with different scores that the integrator can then use to make the decision to adopt the FLOSS product.

5.3 Modularity as the enabler to FLOSS quality

FLOSS's paradox of having and adding more developers without compromising its productivity (in contrast to Brook's law that says that adding more developers increases coordination costs and decreases performance) is due to its approach to modularity. A FLOSS project is made up of many subprojects where only a few developers work together without ever having to interact with the developers in other subprojects or modules [69].

It is believed this is the reason that projects such as Linux and Apache are considered successful. They have been able to scale because of their modularity. Because of modularity, defects in one module, do not affect the rest [58].

However, there is no single definition of modularity. The studies that have defined and measured it do so differently. One study used "correlation between functions added and functions modified" to measure modularity. It then compared modularity across a set of open and closed projects. The open projects did not prove to be more modular than the closed ones [2].

Another study used average component size, which was measured as program length (sum of the number of unique operands and operators) divided by number of statements. The study found that applications with smaller average component size received better user satisfaction scores [1].

In terms of influencing quality, modularity has produced mixed results. A study found that higher modularity does not lead to higher quality. This study defined modularity as the distance of each package in a release from the main sequence. Because higher modularity is associated with reduced software complexity, it should result in higher structural code quality, but the authors found that the projects with higher modularity contained the greater number of defects [37].

Yet another study contests that it is small component design that leads to low defect density, higher user satisfaction, and easier maintenance and evolution [55].

Work distribution is another way of conceptualizing modularity. In another study, the authors found that a lower concentration of developers making changes to a module led to higher quality for the module. The authors speculated that this could explain the FLOSS paradox of many developers and high productivity: at the project level there could be many developers, but within the project, they should be organized into small teams; this would keep the concentration of authors to code low, thus fostering simpler code, higher quality and better maintainability [41].

The literature has defined and operationalized modularity differently using either software structure measures (component size, distance from main sequence, etc.) or development organizational measures such as author concentration per class, number of authors per module, etc.

The development organizational measures have proven to be more effective at finding a correlation between quality and modularity, but these measures are still vaguely defined and more research needs to be performed to optimally define them and operationalize them in order to produce a universal measure of modularity.

5.4 Characteristics of samples

The researchers seem to have made very arbitrary choices when it came to choosing FLOSS project to make up their samples.

By far, the most popular place to obtain data for FLOSS quality studies is the SourceForge repository maintained by Notre Dame University. But it was not the only place to find data; some case studies concentrated on popular projects that are not hosted by SourceForge such as Apache [14], [58], Linux [42], and Eclipse [46].

Some projects considered software type a defining factor and only looked at projects of the same type [40], [41], [42], [62], while the rest did not consider it a factor. However one study did consider it and found that project category affects the bug resolution time [47].

The development time of the projects examined by the authors was extremely variable. There were studies that examined FLOSS project data that covered development time for one week [48], 105 weeks [51], four months [56], six months [46], etc. with one project capturing data from initial commit until the last commit before the stable version was released [41].

Another factor that varied across the studies was the number of projects examined. From four [48], to 52 [49], all the way to 140 [61], and beyond.

Another factor used to choose candidate was the success of the project measured in popularity terms such as number of downloads [61] and SourceForge rank [43], [68], which uses number of downloads and recommendations. Researchers refrained from including *failed* projects and only looked at those that high success measures.

The way researchers are choosing their samples is definitely a reason why there are mixed results in the FLOSS quality literature. They are looking at different types of projects, examining them for different amounts of time, and only considering popular projects.

5.5 Summary

Quality is very subjective and hard to define absolutely. With this challenge, FLOSS researchers have used many ways to define quality. They have used product and process metrics and have found mixed results. FLOSS software is always evolving and one version might produce more defects than a pervious one because of some major change in the software or the community structure.

Successful projects are those that have adopted a modular organization of their code and their community, allowing them to grow and isolate defects. They have also implemented tools to automate policy enforcement and adapted traditional software development practices to their context.

There is a need to evaluate the quality of FLOSS projects, and maturity assessment models have emerged to meet this need. However, they are hard to automate, and their scores are hard to interpret.

An important reason as to why researchers have obtained mixed results in researching FLOSS quality is that their samples have different characteristics in terms of number of projects examined, software type, time evaluated, and popularity of projects examined.

6 Discussion

The reviewed papers show that there is a need to define and quantify quality in FLOSS development projects in order to compare them among each other and to traditionally developed software. Identifying projects that produce high quality products will lead to further research into understanding the factors that lead to higher quality and the interaction of those factors in FLOSS development projects.

The development of assessment models to ascertain quality comes from the position in traditionally developed software that established and repeatable processes lead to the development of quality products.

6.1 FLOSS Quality as evolving

With each release, the FLOSS software and its community change. Quality is not linear: the tenth release of a software product might not have fewer defects than its first. It all depends on what type of release it is; whether it is adding new features, restructuring the entire product, restructuring the way it is developed, or simply posting defect fixes. Which type of release of the product is a more important determinant of its quality than its software structure, or its number of developers.

This explains the mixed results obtained from research that only used product measures as a measure of quality – the modules with the highest change rate and the highest number of defects were not those with the lowest design quality or complexity [38].

6.2 Quality as Defect Resolution Rate

Number of defects added by a release divided by the number of lines of code added by the release would seem a good measure of software product quality [58] that would allow comparison between open and closed software products.

But this assumption is wrong because the release of a commercial closed software product is not the same as the one from an open source project. An open source product release resembles the commercial software after its feature test, since there is neither system nor regression testing in open source projects [58].

These measures do not take into account FLOSS projects' community development. That is why a better measure of FLOSS quality is defect resolution rate, in terms of number of bugs resolved and average time of bug resolution. These measures not only show the quality of the code but also the community's effectiveness at achieving quality.

A key issue is to define bugs as defects in the software product. Bug databases, which are used to calculate the defect resolution rates, are riddled with non-bugs, which must not be taken into account when calculating these rates.

6.3 Modularity as Driver of Quality

The "many eyeballs" looking at the bugs include core developers, periphery developers, sometime contributors, and users, who can easily find their way to the project's publicly available bug tracking system. This group has activity rates, contribution amounts, contributions included per release, problem reports contributed, problem reports resolved, and download statistics. These are all metrics of the community's quality efforts.

But making sure that these community members can work effectively with each other is very necessary. A modular architecture of the code and the community allows a project to grow and attract new developers without having the defects of one group affect another group.

That is why modularity needs to be defined in terms of technical modularity (the coupling of the modules) and organizational modularity (the coupling of the module managers/owners and the core project manager) [70].

However, a downside of modularity is that if a member leaves, his or her module might become orphaned. That is why projects such as Debian are developing their own "quality assurance" groups (http://qa.debian.org), where anyone interested can join and help with mass bug filing and transitions, track orphan code, etc.

It seems that too much modularity might be bad for quality in the long run. It is important to do more research to understand what the right amount of modularity looks like.

6.4 Process and Product as Drivers of Quality

Product and process requirements, the traditional specific quality requirements [71] are still relevant in driving quality in FLOSS products: they are universally understood and any project community still needs to reach for them. But because of this, they are sometimes taken for granted; their inclusion needs to be aided by automation tools, such as testing tools included in the automatic build software that calculates and posts correctness and reliability metrics and compares them with benchmark numbers, alerting the community members if their software product falls below the thresholds.

Relying on the "many eyeballs" to report and fix defects has helped FLOSS achieve quality, but there is something to be said for automating the process in order to produce a higher quality product before it is released.

6.5 FLOSS requires its own Maturity Model for Quality

The development of maturity models such as QualOSS, QSOS, OpenBRR, shows the need for a process evaluation model like CMMI but for FLOSS.

This means that quality could also be defined in terms of this process maturity model, but for this approach to reach maturity (so that one day we might have *level 5* FLOSS projects) more research needs to occur to define, if not the ideal, the most effect FLOSS development processes.

7 Conclusion

Just like in traditionally developed software, there is little consensus in the FLOSS literature when it comes to defining quality.

Linux and Apache are by far the most studied projects in FLOSS literature. All the reviewed papers studied projects that they considered successful: they had released several versions, and had high popularity rating, and download numbers. However, failed projects also need to be studied in order to determine what led to their downfall.

FLOSS communities and their software product are emergent and need to a measure of quality that will reflect their nature. Defect resolution rates (amount of defects resolved, speed of resolution) are the best way to measure a community's commitment to quality, because they recognize that FLOSS is not a static product, but ever evolving. These rates should be calculated per release, and not cumulatively, because the cycle of FLOSS evolution is the release. Researchers should be careful to only include defects and not new feature requests, duplicates, or poorly reported bugs into their calculations.

Modularity is being touted as the main driver of FLOSS quality success, but it needs to be further defined and studied in order to understand how it works.

References

1. Stamelos, I., Angelis, L., Oikonomou, A., Bleris, G.L.: Code Quality Analysis in Open Source Software Development. Information Systems Journal 12, 43-60 (2002)
2. Paulson, J.W., Succi, G., Eberlein, A.: An empirical study of open-source and closed-source software products. Software Engineering, IEEE Transactions on 30, 246-256 (2004)
3. Kuan, J.: Open Source Software as Lead-User's Make or Buy Decision: A Study of Open and Closed Source Quality. Second Conference on The Economics of the Software and Internet Industries, (2003)
4. Raghunathan, S., Prasad, A., Mishra, B.K., Chang;, H.: Open source versus closed source: software quality in monopoly and competitive markets. IEEE Transactions on Systems, Man and Cybernetics, Part A, 35, 903-918 (2005)
5. Crowston, K., Wei, K., Howison, J., Wiggins, A.: Free/Libre Open Source Software Development: What We Know and What We Do Not Know. ACM Computing Surveys 44, (2012)

6. Ghosh, R.A.: Economic Impact of Open Source Software on Innovation and the Competitiveness of the Information and Communication Technologies Sector in the E. U. . (2006)
7. von Hippel, E., von Krogh, G.: Open Source Software and the "Private-Collective" Innovation Model: Issues for Organization Science. Organization Science 14, 209-223 (2003)
8. Hales, P.: Firefox use continues to rise in Europe. The Inquirer, (2006)
9. Scacchi, W.: Understanding Requirements for Open Source Software. In: Lyytinen, K., Loucopoulos, P., Mylopoulos, J., Robinson, W. (eds.) Design Requirements Engineering - A Multi-Disciplinary Perspective for the Next Decade. Springer-Verlag (2009)
10. Stewart, K.J., Gosain, S.: The Impact of Ideology on Effectiveness in Open Source Software Development Teams. MIS Quarterly 30, 291-314 (2006)
11. Ajila, S.A., Wu, D.: Empirical study of the effects of open source adoption on software development economics. Journal of Systems and Software 80, 1517-1529 (2007)
12. Bonaccorsi, A., Rossi, C.: Comparing Motivations of Individual Programmers and Firms to Take Part in the Open Source Movement. Knowledge, Technology and Policy 18, 40-64 (2006)
13. Raymond, E.: The cathedral and the bazaar. Knowledge, Technology, and Policy 12, 23-49 (1999)
14. Rigby, P.C., German, D.M., Storey, M.-A.: Open source software peer review practices: a case study of the apache server. 0th International Conference on Software Engineering (ICSE2008), pp. 541-550, Leipzig, Germany (2008)
15. The Open Source Initiative, http://www.opensource.org/docs/osd
16. Ye, Y., Kishida, K.: Toward an Understanding of the Motivation of Open Source Software Developers. Proceedings of the 25th International Conference on Software Engineering, pp. 419-429 (2003)
17. Mockus, A., Fielding, R.T., Herbsleb, J.D.: Two case studies of open source software development: Apache and Mozilla. ACM Trans. Softw. Eng. Methodol. 11, 309-346 (2002)
18. Schwaber, K.: Agile Project Management with Scrum. Microsoft Press (2004)
19. Deming, W.E.: Quality, Productivity, and Competitive Position. MIT Center for Advanced Engineering Study, Cambridge, MA, USA (1982)
20. Deming, W.E.: Out of the Crisis. MIT Center for Advanced Engineering Study, Cambridge, MA, USA (1986)
21. Juran, J.M.: Planning for Quality. Collier Macmillan, London, UK (1988)
22. Crosby, P.B.: Quality is Free: The Art of Making Quality Certain. McGraw-Hill (1979)
23. Feigenbaum, A.: Total quality control: engineering and management: the technical and managerial field for improving product quality, including its reliability, and for reducing operating costs and losses. McGraw-Hill (1961)
24. Ishikawa, K.: What is total quality control? The Japanese way. Prentice-Hall (1985)
25. Tennant, G.: Six Sigma: SPC and TQM in manufacturing and services. Gower Publishing (2001)
26. Garvin, D.A.: What does 'Product Quality' really mean? Sloan Management Review 1, 25-43 (1984)
27. Boehm, B.W., Brown, J.R., Lipow, M.: Quantitative evaluation of software quality. Proceedings of the 2nd international conference on Software engineering. IEEE Computer Society Press, San Francisco, California, United States (1976)
28. Cavano, J.P., McCall, J.A.: A Framework for the Measurement of Software Quality. Proceedings of the ACM Software Quality Workshop, pp. 133-139. ACM (1978)
29. McCall, J.A., Richards, P.K., Walters, G.F.: Factors in Software Quality. National Technology Information Service Vol 1, 2, and 3, (1977)
30. ISO: ISO 9126-1:2001, Software engineering - Product quality, Part 1: Quality model. (2001)

31. Walsham, G.: The Emergence of Interpretivism in IS Research. Information Systems Research 6, 376-394 (1995)
32. Strauss, A.L., Corbin, J.M.: Basics of Qualitative Research: Grounded Theory Procedures and Techniques. Sage Publications, Newbury Park, California, U.S.A. (1990)
33. Strauss, A., Corbin, J.: Grounded Theory Methodology - An Overview. In: Denzin, N.K., Lincoln, Y.S. (eds.) Handbook of Qualitative Research, pp. 273-285. Sage Publications, Thousand Oaks, CA (1994)
34. Aksulu, A., Wade, M.: A Comprehensive Review and Synthesis of Open Source Research. Journal of the Association for Information Systems 11, 576-656 (2010)
35. Spinellis, D.: A Tale of Four Kernels. 30th International Conference on Software Engineering, 2008. ICSE '08, pp. 381-390. ACM/IEEE, Leipzig, Germany (2008)
36. Capra, E., Francalanci, C., Merlo, F.: An Empirical Study on the Relationship among Software Design Quality, Development Effort, and Governance in Open Source Projects. IEEE Transactions on Software Engineering 34, 765-782 (2008)
37. Conley, C.A.: Design for quality: The case of Open Source Software Development. Stern Graduate School of Business Administration, vol. PhD, pp. 43. New York University, New York, NY, USA (2008)
38. Koru, A.G., Tian, J.: Comparing high-change modules and modules with the highest measurement values in two large-scale open-source products. Software Engineering, IEEE Transactions on 31, 625-642 (2005)
39. Gyimothy, T., Ferenc, R., Siket, I.: Empirical validation of object-oriented metrics on open source software for fault prediction. Software Engineering, IEEE Transactions on 31, 897-910 (2005)
40. Koru, A.G., Liu, H.: Identifying and characterizing change-prone classes in two large-scale open-source products. Journal of Systems and Software 80, 63-73 (2007)
41. Koch, S., Neumann, C.: Exploring the Effects of Process Characteristics on Product Quality in Open Source Software Development. Journal of Database Management 19, 31-57 (2008)
42. Yu, L., Schach, S.R., Chen, K., Heller, G.Z., Offutt, J.: Maintainability of the kernels of open-source operating systems: A comparison of Linux with FreeBSD, NetBSD, and OpenBSD. Journal of Systems and Software 79, 807-815 (2006)
43. Barbagallo, D., Francalenei, C., Merlo, F.: The Impact of Social Networking on Software Design Quality and Development Effort in Open Source Projects. Proceedings of the International Conference on Information Systems, (2008)
44. Samoladas, I., Stamelos, I., Angelis, L., Oikonomou, A.: Open source software development should strive for even greater code maintainability. Commun. ACM 47, 83-87 (2004)
45. Ghapanchi, A.H., Aurum, A.: Measuring the Effectiveness of the Defect-Fixing Process in Open Source Software Projects. Proceedings of the 44th Hawaii International Conference on System Sciences, Hawaii, USA (2011)
46. Kidane, Y., Gloor, P.: Correlating temporal communication patterns of the Eclipse open source community with performance and creativity. Computational & Mathematical Organization Theory 13, 17-27 (2007)
47. Au, Y.A., Carpenter, D., Chen, X., Clark, J.G.: Virtual organizational learning in open source software development projects. Information & Management 46, 9-15 (2009)
48. Crowston, K., Scozzi, B.: Bug fixing practices within free/libre open source software development teams. Journal of Database Management 19, 1-30 (2008)
49. Koru, A.G., Tian, J.: Defect handling in medium and large open source projects. Software, IEEE 21, 54-61 (2004)
50. Glance, D.G.: Release Criteria for the Linux Kernel. First Monday 9, (2004)
51. Huntley, C.L.: Organizational learning in open-source software projects: an analysis of debugging data. Engineering Management, IEEE Transactions on 50, 485-493 (2003)

52. Sohn, S.Y., Mok, M.S.: A strategic analysis for successful open source software utilization based on a structural equation model. Journal of Systems and Software 81, 1014-1024 (2008)
53. Zhou, Y., Davis, J.: Open source software reliability model: an empirical approach. Proceedings of the fifth workshop on Open source software engineering, pp. 1-6. ACM, St. Louis, Missouri (2005)
54. Zhao, L., Elbaum, S.: Quality assurance under the open source development model. Journal of Systems and Software 66, 65-75 (2003)
55. Aberdour, M.: Achieving Quality in Open Source Software. IEEE Software 24, 58-64 (2007)
56. Halloran, T.J., Scherlis, W.L.: High Quality and Open Source Software Practices. Proceedings of the 2nd Workshop on Open Source Software Engineering (ICSE 2002), Orlando, FL, USA (2002)
57. Michlmayr, M., Hunt, F., Probert, D.: Quality Practices and Problems in Free Software Projects. In: Scotto, M., Succi, G. (eds.) Proceedings of the First International Conference on Open Source Systems, pp. 24-28, Genova, Italy (2005)
58. Mockus, A., Fielding, R.T., Herbsleb, J.: A Case Study of Open Source Software Development: The Apache Server. Proceedings of the 22nd International Conference on Software Engineering (ICSE), (2000)
59. Samoladas, I., Gousios, G., Spinellis, D., Stamelos, I.: The SQO-OSS Quality Model: Measurement Based Open Source Software Evaluation. 4th International Conference on Open Source Systems (OSS2008), pp. 237-248, Milan, Italy (2008)
60. Deprez, J.-C., Alexandre, S.: Comparing Assessment Methodologies for Free/Open Source Software: OpenBRR and QSOS. In: Jedlitschka, A., Salo, O. (eds.) PROFES 2008, pp. 189-203 (2008)
61. Crowston, K., Howison, J., Annabi, H.: Information Systems Success in Free and Open Source Software Development: Theory and Measures. Software Process: Improvement and Practice 11, 123-148 (2006)
62. Wray, B., Mathieu, R.: Evaluating the performance of open source software projects using data envelopment analysis. Information Management & Computer Security 16, 449 (2008)
63. del Bianco, V., Lavazza, L., Morasca, S., Taibi, D., Tosi, D.: The QualiSPo approach to OSS product quality evaluation. Proceedings of the 3rd International Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development (FLOSS '10), pp. 23-28. ACM, Cape Town, South Africa (2010)
64. Soto, M., Ciolkowski, M.: The QualOSS open source assessment model measuring the performance of open source communities. Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement, pp. 498-501 (2009)
65. Deprez, J.-c., Monfils, F.F., Ciolkowski, M., Soto, M.: Defining Software Evolvability from a Free/Open-Source Software Perspective. Third International IEEE Workshop on Software Evolvability, pp. 29-35. IEEE, Paris, France (2007)
66. Glott, R., Groven, A.-K., Haaland, K., Tannenberg, A.: Quality Models for Free/Libre Open Source Software--Towards the "Silver Bullet"? 36th EUROMICRO Conference on Software Engineering and Advanced Applications, pp. 439-446, Lille, France (2010)
67. Groven, A.-K., Haaland, K., Glott, R., Tannenberg, A.: Security measurements within the framework of quality assessment models for free/libre open source software. Proceedings of the Fourth European Conference on Software Architecture: Companion Volume, pp. 229-235. ACM, Copenhagen, Denmark (2010)
68. Michlmayr, M.: Software Process Maturity and the Success of Free Software Projects. Proceeding of the 2005 conference on Software Engineering: Evolution and Emerging Technologies, pp. 3-14 (2005)

69. Schweik, C.M., English, R.C., Kitsing, M., Haire, S.: Brooks' Versus Linus' Law: An Empirical Test of Open Source Projects. Proceedings of the 2008 international conference on Digital government research, pp. 423-424, Montreal, Canada (2008)
70. Tiwana, A.: The Influence of Software Platform Modularity on Platform Abandonment: An Empirical Study of Firefox Extension Developers. University of Georgia, Terry School of Business (2010)
71. Glinz, M.: On Non-Functional Requirements. Proceedings of the 15th IEEE International Requirements Engineering Conference, vol. 0, pp. 21-26, Delhi, India (2007)