An Exploration of Code Quality in FOSS Projects

Iftekhar Ahmed, Soroush Ghorashi , and Carlos Jensen
School of Electrical Engineering and Computer Science
Oregon State University, Corvallis, OR, USA
ahmed@eecs.oregonstate.edu, ghorashs@onid.oregonstate.edu, cjensen@eecs.oregonstate.edu

Abstract. It is a widely held belief that Free/Open Source Software (FOSS) development leads to the creation of software with the same, if not higher quality compared to that created using proprietary software development models. However there is little research on evaluating the quality of FOSS code, and the impact of project characteristics such as age, number of core developers, code-base size, etc. In this exploratory study, we examined 110 FOSS projects, measuring the quality of the code and architectural design using code smells. We found that, contrary to our expectations, the overall quality of the code is not affected by the size of the code base, but that it was negatively impacted by the growth of the number of code contributors. Our results also show that projects with more core developers don't necessarily have better code quality.

Keywords: Code Quality, Success Metrics, FOSS, Open Source Software.

1 Introduction

Free/Open Source Software (FOSS) is associated with collaborative development model bringing developers together from different geographical locations to create cost effective and efficient software. The adoption of FOSS is growing; Walli et al. found that, out of the 512 U.S. companies surveyed, 87% of them used FOSS [43]. Another survey in 2007 also found that not only does FOSS have a significant market share, but that this development model in many cases produces the more reliable and highest performing software option [45].

The quality of FOSS software has been subject to debate. Though some studies have argued for that FOSS projects can produce high-quality code [33, 36, 6], critics often point to the lack of formal project management practices and requirements as roadblocks to reliably achieving high software quality [35]. That said, the FOSS community has developed its own methods of quality assurance and quality control [33], and researchers have shown that FOSS projects produce more secure code due to peer-review and the openness of the code [15]. Despite this controversy, there is a lack of large-scale empirical studies of objective code quality .

Evaluating the quality of software can be a difficult task because there are a large number of properties that could be evaluated [16, 44], such as functionality, adherence to specifications, security, usability, etc. [18, 6]. It is not clear that these factors, even when combined, would give an adequate definition of quality. As a result many researchers have used different project characteristics as a substitute measure of the quality of a project, including longevity [13], operational software characteristics [31], number of open bugs [5], etc. None of these properties measure code quality directly (e.g. open bug reports is at best an indirect measure, as it is confounded by the size of the code base and the quality and extent of testing). One could argue that in the absence of requirements and objective definitions of quality, the more objective measure to focus on would be the quality of the code itself.

We decided to examine the quality of FOSS source code using two types of code smells: implementation level code smells and design level code smells, two objective measures of code quality. A code smell is not an error or problem in itself, but rather a set of heuristics for identifying poor coding practices or code structure, often associated with poor maintainability, bugs, or inefficiencies [19]. This definition refocuses code quality to the maintainability and efficiency of the code.

We used existing code-smell detection tools to analyze implementation level code smells (focusing on violations of standard coding practices), and design level smells (focusing on higher-level issues associated with architectural decisions), and see if there were interactions between these and other project characteristics. In this paper we sought to answer the following questions:

- What impact do project characteristics such as longevity, size of code base and number of developers have on the low-level quality of the code?
- What impact do project characteristics such as longevity, size of code base and number of developers have on the quality of design-level decisions?

The outline of this paper is as follows. We review related work in FOSS development model. Next we describe our research methodology. Then we present our findings followed by the discussion, and finally outline future work and conclusion.

2 Background

Two of most important characteristics of the FOSS development model, for the purposes of this paper, are the distributed development process and the reliance on unpaid developers. These two factors can have an important effect on software quality because they lead to decentralization and difficulty compelling developers to take specific actions. That said, these characteristics are not indicative of a lack of quality; there are plenty of examples of outstanding FOSS projects, such as Linux, Apache, Gnome, Mozilla, etc.

Most FOSS research is focused on these successful and popular projects [27, 34], while ignoring the many smaller FOSS projects that fail to reach this level of quality, popularity, or success. While a core reason for failure can be a lack of interest or volunteers [40], some suggest that lack of management is a major cause [38].

One key to the success of FOSS is that source code availability allows faster evolution and higher quality because of parallel development and the theory that having more eyes on the code identifies more bugs quickly [32]. Because of this, as Stamelos et al. [39] found, mature FOSS code is generally of good quality.

Many metrics have been used to define the success of an information system (IS) such as a FOSS project. Delone and McLean's IS success model is perhaps the most cited [8, 9, 7]. In this model they identify six measures of success, with system quality being the most important measure for user satisfaction and adoption [22].

Looking beyond FOSS, Boehm et al. [3] and Gorton and Liu [14] among many software engineering researchers, have explored different measures for software quality; including completeness, usability, testability, maintainability, reliability, efficiency and etc. that are all relevant to our analysis.

Code smells are symptoms of problems in the source code, and an indicator of where refactoring is needed [11]. This in turn has been associated with errors [23] and code maintainability problems [11]. Researchers have come up with different types of code smells depending on their type of impact [25]. The identification of code smells is typically done during development, testing, and maintenance.

Many approaches have been proposed for code smell detection, such as metric based [20] and meta-model based [28]. Metric based measures show that code smells impact software quality [24]. Most of the code smell detection tools are based on metrics analysis [20]. This static analysis based approach has its drawbacks. Fowler and Beck claimed that *"No set of metrics rivals informed human intuition"* [11] when it comes to deciding whether an instance of a code smell should be refactored.

3 Methodology

Our goal was to analyze the impact of different project characteristics on the overall quality of code, measured using coding practice violations and design level code smells. We wanted to see if project characteristics such as longevity, number of core developers, the size of the developer community, the frequency of code changes, and size of the code base were correlated with the quality of the resulting code. This could shed valuable light on prevalent assumptions about the evolution of FOSS projects, and prescribed best practices.

We sought to perform an analysis of representative FOSS projects, but we also wanted to control for as many external factors as possible. One such factor was the programming language used. We decided to restrict our study to Java for two reasons: First, Java is one of the most popular languages (according to Github [12] and the Tiobe index [41]). Second, the number and robustness of Source Code Analysis (SCA) and code smell detection tools for Java compared to other languages.

For code smell detection we built on the work of Fontana et al [10] and selected the InFusion [17] toolset. To identify the standard coding practice violations, we used "Codepro Analytix" [4] which uses a set of 643 rules collected from textbooks of java such as *The Elements of Java Style* [42] and *Effective java* [2].

We used the 110 most popular Java projects hosted on Github [12]. Popularity was measured using the number of stars given to projects, a product of the number of users who liked the project and chose to get updates about it. Core developers in this paper are defined as the group of contributors that contribute major portions of the commits and also act as reviewers for patches submitted by others. Table 1 gives an overview of the key characteristics of the projects in our dataset.

Table 1. Summary of the project characteristics

	Min	Max	Median	Average	Stddev
File count	480	462,846	22,811	51,270.0	74,354.5
LOC	701	968,287	45,564	10,702.1	175,146.2
# of commits	20	120,947	555	2,880.7	11,891.5
Age (days)	5	5,485	910	1,086.0	905.8
# of core developers	1	44	3	6.1	7.6
# of contributors	1	225	25	43.7	47.2
Popularity (stars)	695	7,252	1,011	1,376.5	987.8
Total design code smells	0	4,981	28	231.7	645.3
Total coding violations	18	4,894	921	1,074.1	908.9

In the second phase we analyzed the projects to check if project characteristics contributed towards specific design issues. We categorized code smells into broader categories, as suggested by the code smell literature [25]. Our categories were: Bloater, Object oriented abusers, Coupler, Dispensables, Encapsulators and Others.

- *Bloaters* are smells that leads the code to balloon so that it cannot be effectively handled. This includes data clumps, large class, long method, long parameter list and primitive obsession [25].
- *Object oriented abusers* are smells that do not fully exploit the advantages of object-oriented design. Some of the smells include Switch statements, parallel inheritance hierarchies, and alternative classes with different interfaces [29].
- The *Coupler* category contains smells related to high coupling between objects, in defiance of good object oriented design principles. Smells in this category include feature envy and inappropriate intimacy [29].
- The *Dispensable* category contains smells such as the lazy class, data class and duplicate codes [29].
- The *Encapsulators* category contains smells that deal with data communication or encapsulation, and includes message chain and middleman smells [29].
- *Others* is an aptly named catch-all category.

Two researchers independently performed this categorization. Inter-rater agreement was calculated using Cohen's Kappa. As the categorization was straightforward and there were only 6 categories, we achieved a Cohen-Kappa score of 0.90 after the first round of coding. According to Landis and Koch [19], Cohen-Kappa score greater than .85 indicates almost perfect agreement between coders.

4 Results

We calculated correlation coefficient between the number of low-level smells and the project characteristics mentioned earlier. None of these showed a Pearson

Correlation Coefficient greater than 0.50, meaning that there was no strong linear correlation between any of the single properties and overall low-level code quality.

Next we used linear regression analysis with the intercept β_0 set to zero (absence of files or other variables results in zero code violations). Our model discarded Popularity and Number of commits as significant factors (see Table 2).

Table 2. Quality indicator of the linear model for measuring low-level code smells

Title	Estimate	Std. Error	t value
File count	-0.4401	0.1996	-2.205
LOC	0.3233	0.1549	2.087
# of core developers	-0.6321	0.2715	-2.328
# of contributors	0.8408	0.2020	4.162
Age (Days)	0.4563	0.1506	3.030
***R-Squared**	**0.4735**		

To answer our second research question we checked for correlations between the number of design level smells and the project characteristics. Surprisingly we found that there was a strong liner relationship between the number of design level code smells and the number of core developers (Pearson Correlation Coefficient = 0. 73).

Next we used linear regression analysis with the intercept β_0 set to zero (absence of files or other variables results in zero code violations). Our model discarded LOC, Number of contributors and Popularity as significant factors (see Table 3).

Table 3. Quality indicator of the linear model for measuring design level code smells

Title	Estimate	Std. Error	t value
File count	0.14303	0.10169	1.406
# of core developers	0.49993	0.08675	5.763
# of commits	0.16490	0.7962	2.071
Age (Days)	-0.13961	0.08079	-1.728
# of lines deleted	0.13450	0.07667	1.754
***R-Squared**	**0.5349**		

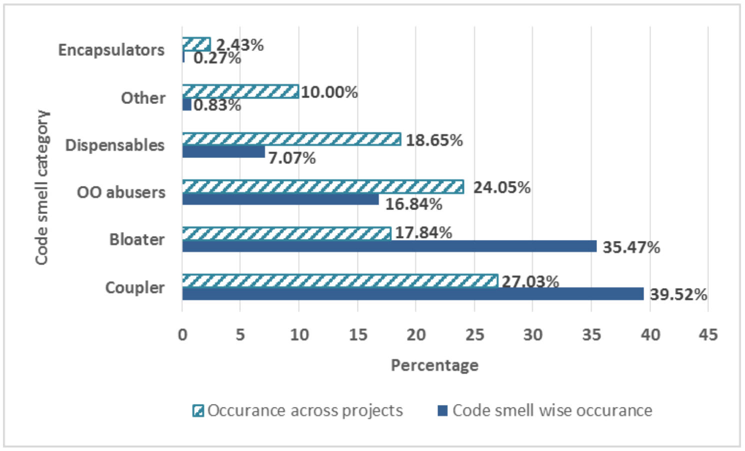

Fig. 1. Percentage occurrence of code smell categories

Figure 1 shows the mapping of design-level code smells into the six high-level categories discussed in our methodology section. We found that Couplers and Bloaters are the most common design-oriented code smells (solid bars), with 39.52% and 35.47% of all occurrences respectively. However, when we look at the percentage of projects that have at least one of these smells we see a much more even distribution (striped bars).

5 Discussion

The linear regression models that we found have R-squared values of 0.4735 and 0.5349 for low-level smells and design-oriented smells respectively. This means that a handful of project factors can account for roughly 50% of the variance in the sample. We also found that for the design smells, there was a strong correlation between the number of core developers and smells.

Looking at the data we see few interesting things standing out. For low level smells we were not surprised to see a correlation between the size of the code base or the number of contributors on one hand, and the number of smells on the other. This was expected; the more complex the code and the more coders there are, the harder it is to curate the code effectively, thus allowing more code smells to manifest. We also expected file count and number of core developers to have a positive effect on code quality, as the first leads to better organization, and the second better curating and supervision.

What was somewhat surprising was to see that the age of the code-base had a negative effect on code quality. One would expect the code to improve over time, but it appears as if this is not necessarily the case. It is likely that the urge to add new features, or the turnover of programmers outweigh any refactoring and fine-tuning performed by the community.

For high-level, or design oriented smells, we found that file count, number of commits and number of core developers actually led to an increase in the number of smells. Most of these correlations make sense. While the number of files help manage the low-level complexities of the code, adding files makes it harder to maintain the high-level conceptual design. Likewise, while adding core developers help projects stay on top of low-level code reviews, but hinders the high-level design vision. Too many cooks do seem to spoil he broth.

Unexpectedly we found that age did decrease the number of design smells, which means that though age is associated with more low-level smells, it is also associated with fewer design smells. It may be that over time, refactoring is more often aimed at removing design flaws rather than low-level coding convention violations.

Turning to the last part of our analysis, we find support for our interpretation of the results; the code smell categorization gives us a deeper understanding of the underlying causes for code smells. Coupler was the most common code smell, and represents the smells that causes high coupling between objects [29]. This might be an indication of a lack of adherence to object oriented design principles. We can hypothesize that, this type of ad hoc change happens due to the lack of knowledge about the design decisions of the project and due to low adoption rate of modeling and design tools in FOSS community [33].

Bloaters were the second most common code smell, and represent smells that lead the code to grow out of control, and is often caused by small changes and additions to the code [29]. Bloating is associated with centralized control structures using object-oriented languages, and Arisholm et al. identified that novice developers perform better with centralized control styles [1]. So it's most likely that novice developers turned contributors are pushing these changes. This potentially raises a question about quality assurance in FOSS.

In our analysis we have used the total number of coding standard violations and code smells as the indicator of FOSS projects quality. Identified regression models indicate that the number of core developers is correlated with quality of the FOSS project. This is in line with the findings of Sen et al., who found that number of developers reflect the "healthiness" of a FOSS project [37]. Contrary to one of the findings of Sen et al. we didn't find any strong relationship between the popularity of a project and the quality of the project. This can be explained by the "lurking" phenomenon that is prevalent in online communities [30] up to 90% [26]. The "lurkers" are members of the community, who do not participate in any activity except watching, so it make sense that they have little impact on the actual quality of the code. Further empirical analysis is required to identify the actual contribution of lurking towards this observation to make any concluding remarks on this topic.

6 Limitations

It's always difficult to generalize a diverse movement such as FOSS. While we tried to ensure diversity amongst the projects we selected, there were limitations to our methodology and selection criteria. We list the most important here:

Our analysis was of 110 FOSS projects, which though a reasonable sample, is small compared to a population of 375,486 projects. Though we selected the projects based on popularity we did not consider the application domain of the selected projects. During the data collection, we also did not exclude folders that were not directly related to the functionality of the system itself (test folders, contribution folders, demo folders, etc.).

7 Conclusion

We were able to show that there is a correlation between a number of important factors such as the longevity, the number of developers, and code-base size on one side, and the number of low-level code smells on the other. This implies that as projects grows and ages, the quality of the code decreases, unless counteracted by a larger group of core contributors to curate submissions. However, we also found that increasing the number of core developers negatively affects the higher-level design of the code. It is therefore important to carefully balance the size of the core group.

Acknowledgements

We would like to thank the Oregon State University HCI group for their input and feedback on the research.

References

1. Arisholm, E., & Sjoberg, D. I. (2004). “Evaluating the effect of a delegated versus centralized control style on the maintainability of object-oriented software”. In *IEEE Transactions on Software Engineering*, 30(8), (pp.521-534).
2. Bloch, J. (2008). “Effective java. Addison-Wesley Professional”.
3. Boehm, B. W., Brown, J. R., & Lipow, M. (1976). “Quantitative evaluation of software quality”. In Proc. *2nd international conference on Software engineering*, (pp. 592-605).
4. CodePro Analytix, https://developers.google.com/java-dev-tools/codepro/doc/
5. De Groot, A., Kügler, S., Adams, P. J., & Gousios, G. (2006). “Call for quality: Open source software quality observation.” In Proc.*Open Source Systems*, (pp. 57-62). Springer US.
6. Del Bianco, V., Lavazza, L., Morasca, S., Taibi, D., & Tosi, D. (2010). “An Investigation of the users’ perception of OSS quality”. In *Open Source Software*: *New Horizons* (pp. 15-28). Springer Berlin Heidelberg.
7. DeLone, W. H., & McLean, E. R. (2002). “Information Systems Success Revisited,” In Proc. of the *35th Hawaii International Conference on System Sciences*.
8. DeLone, W. H., & McLean, E. R. (2003) “The DeLone and McLean Model of Information Systems Success: A Ten-Year Update,” In *Journal of Management Information Systems*, (pp. 9-30).
9. DeLone, W. H., & McLean, E. R “Information Systems Success: The Quest for the Dependent Variable,” In *Information Systems Research*, (pp. 60-95).
10. Fontana, F. A., Mariani, E., Morniroli, A., Sormani, R., & Tonello, A. (2011). “An experience report on using code smells detection tools”. In *Software Testing, Verification and Validation Workshops (ICSTW)*, (pp. 450-457).
11. Fowler, M. (1999)." Refactoring: improving the design of existing code. Addison-Wesley Professional".
12. Github, https://github.com
13. Golden, B. (2005)." Making Open Source Ready for the Enterprise, The Open Source Maturity Model". Extracted From *Succeeding with Open Source, Addison-Wesley Publishing Company*

14. Gorton, I., & Liu, A. “Software Component Quality Assessment in Practice: Successes and Practical Impediments,” in Proc. of the *24th International Conference on Software Engineering, IEEE Computer Society*, (pp. 555-558).
15. Hoepman, J. H., & Jacobs, B. (2007). "Increased security through open source". In *Communications of the ACM*, 50(1), (pp.79-83).
16. I. Sommerville, (2001).”Software Engineering”. Essex, England: Pearson Education Limited.
17. InFusion, http://www.intooitus.com/inFusion.html
18. Jung, H. W., Kim, S. G., & Chung, C. S. (2004)." Measuring software product quality: A survey of ISO/IEC 9126". In *Software, IEEE, 21(5)*, (pp.88-92).
19. Landis, J. R., & Koch, G. G. (1977). “The measurement of observer agreement for categorical data”. In *Biometrics*, 33, (pp.159-174).
20. Lanza, M., & Marinescu, R. (2006). “Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems”. Reading: Springer.
21. Lavazza, L., Morasca, S., Taibi, D., & Tosi, D. (2010). “Predicting OSS trustworthiness on the basis of elementary code assessment”. In Proc. of *ACM-IEEE International Symposium on Empirical Software Engineering and Measurement* (p. 36).
22. Lee, S. Y. T., Kim, H. W., & Gupta, S. (2009)." Measuring open source software success". In Proc. *Omega, 37(2)*, (pp.426-438).
23. Li, W., & Shatnawi, R. (2007)." An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution". In *Journal of Systems and Software, 80 (7)*, (pp.1120-1128).
24. Marinescu, R. (2001). “Detecting design flaws via metrics in object-oriented systems”. In *39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems, 2001. TOOLS 39*, (pp.173-182).
25. Marticorena, R., López, C., & Crespo, Y. (2006). “Extending a taxonomy of bad code smells with metrics”. In Proc.*7th ECCOP International Workshop on Object-Oriented Reengineering (WOOR),* (pp. 6).
26. Mason, B. I. (1999). ”Issues in virtual ethnography”. In Proc. of *Ethnographic Studies in Real and Virtual Environments: Inhabited Information Spaces and Connected Communities*. Edinburgh, (pp. 61-69).
27. Mockus, A., Fielding, R. T., & Herbsleb, J. D. (2002). “Two case studies of open source software development: Apache and Mozilla,” In *ACM Transactions on Software Engineering and Methodology, vol. 11*, (pp. 309–346).
28. Moha, N., Rezgui, J., Guéhéneuc, Y. G., Valtchev, P., & El Boussaidi, G. (2008). “Using FCA to suggest refactorings to correct design defects”. In *Concept Lattices and Their Applications* (pp. 269-275). Springer Berlin Heidelberg.
29. Mäntylä, M. (2003). "Bad smells in software-a taxonomy and an empirical study". Helsinki University of Technology.
30. Nonnecke, B., & J. Preece. (2000) "Lurker Demographics: Counting the Silent". In Proc. *CHI 2000*, (pp.73-80)
31. Rating, B.R. “Business readiness rating for open source”. http:// openbrr.org

32. Raymond, E. (1999). The cathedral and the bazaar. Knowledge, Technology & Policy, 12(3), (pp.23-49).
33. Robbins, J. (2005). "Adopting open source software engineering (OSSE) practices by adopting OSSE tools". In *Perspectives on free and open source software*, (pp.245-264).
34. S. Koch & G. Schneider. (2002). "Effort, cooperation and coordination in an open source software project: GNOME," In *Information Systems Journal*, vol. 12, no. 1, (pp. 27–42).
35. Scacchi, W., Feller, J., Fitzgerald, B., Hissam, S., & Lakhani, K. (2006). "Understanding free/open source software development processes". In *Software Process: Improvement and Practice*, 11(2), (pp.95-105).
36. Schmidt, D. C., & Porter, A. (2001). "Leveraging open-source communities to improve the quality & performance of open-source software". In Proc. of the *1st Workshop on Open Source Software Engineering.*
37. Sen, R., Subramaniam, C., & Nelson, M. L. (2011)." Open source software licenses: Strong-copyleft, non-copyleft, or somewhere in between?". In *Decision Support Systems*, 52(1), (pp.199-206).
38. Senyard, A. and Michlmayr, M. (2004). "How to have a successful free software project," in Proceedings of *the 11th Asia-Pacific Software Engineering Conference. Busan, Korea: IEEE Computer Society*, (pp. 84–91).
39. Stamelos, I., Angelis, L., Oikonomou, A., & Bleris, G. L. (2002). "Code quality analysis in open source software development". *Information Systems Journal,* 12(1), (pp.43-60).
40. Subramaniam, C. (2009)."Determinants of open source software project success: A longitudinal study". In *Decision Support Systems* 46, (pp.576–585)
41. Tiobe, http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html
42. Vermeulen, A. (Ed.). (2000). "The Elements of Java (TM) Style (Vol. 15)". Cambridge University Press.
43. Walli, S., Gynn & D., Rotz, V. (2005). "The Growth of Open Source Software in Organization ", http://dirkriehle.com/wp-content/uploads/2008/03/wp_optaros_oss_usage_in_organizations.pdf
44. Wennergren, D.M. (2009). "Clarifying Guidance Regarding Open Source Software (OSS)", http://dodcio.defense.gov/Portals/0/Documents/FOSS/2009OSS.pdf
45. Wheeler, D. (2007). "Why Open Source Software/Free Software (OSS/FS,FOSS, or FLOSS)? Look at the Numbers!" , http://www.dwheeler.com/oss_fs_why.html