Developer Focus: Lack of Impact on Maintainability

Csaba Faragó[1] and Péter Hegedűs[2]

[1] Department of Software Engineering, University of Szeged, Hungary
`farago@inf.u-szeged.hu`
[2] MTA-SZTE Research Group on Artificial Intelligence, Szeged, Hungary
`hpeter@inf.u-szeged.hu`

Abstract. We were looking for evidence that a connection between source code quality erosion and the developer focus exists. We assumed that more focused developers, i.e. those who are concerned with a well specified part of the source code at a time are likely to commit higher quality code compared to those who are less focused, i.e. committing to various parts of the code. We estimated code quality with the ColumbusQM quality model and developer focus with structural scattering. Despite the assumption sounds quite logical, we could not find any supporting evidence.
As structural scattering assigns a measure to a set of source files/classes (i.e., how close they are to each other in the package hierarchy), we could apply it in various ways. First, we defined developer focus to be the structural scattering of the set of source files in a commit to validate if more focused changes have better impact on maintainability than less focused ones. Second, we calculated the structural scattering of all the files the developer of a commit modified in the last 3 months and assigned this measure as the developer focus to that commit. With this test we checked if more focused developers tend to commit better quality code, compared to less focused ones. We also performed this test for every developer separately, considering only the subset of the commits that were created by that particular developer.
We calculated the level of developer focus and the maintainability changes for every commit of three open-source and one proprietary software system. With the help of Wilcoxon rank test we compared the focus values of commits causing a maintainability increase with those of decreasing the maintainability. The results are non-conclusive, they do not even tend to the same direction, therefore we did not find any evidence of an existing connection between maintainability and developer focus. Therefore this is a publication of negative results.

Keywords: developer focus, ISO/IEC 25010, source code maintainability, Wilcoxon test, negative results

1 Introduction

Maintenance of the software consumes big efforts, high proportion of the total amount of software development costs are spent on this activity. Source code

maintainability is in direct connection with maintenance costs [2]. Our motivation in this work was to investigate the effect of the development process on the maintainability of the code. Our goal was to explore typical patterns causing similar changes in software maintainability, which could either help to avoid software erosion, or provide information about how to better allocate efforts to improve software maintainability.

We already investigated this area of research in previous works [6–11]. In article [11] we showed that a strong connection between the version control operations and the maintainability of the source code exists. In study [7] we revealed the connection between the version control operations and maintainability. We found that file additions have rather positive and file updates have negative effect on maintainability. A clear effect of file deletions was not identified. In article [6] we presented the results of a variance analysis. We found that file additions and file deletions increase the variance of the maintainability, and operation Update decreases it. In work [9] we analyzed code churn, i.e. the intensity of past modifications. We found that modifying high-churn code is more likely to decrease the overall maintainability of a software system. In study [8] we considered the developer and investigated how code ownership impacts maintainability. We concluded that common code is more likely to erode further than code with clear ownership. In article [10] we defined a few version control history metrics and checked their connection with maintainability. Our tests resulted that higher intensity of modifications, the higher number of code modifications and developers, the older code and the later last modification date have lower maintainability and higher number of post-release bugs.

Up to now we published in this area of research positive results only. But we think that publishing negative results is also very important; a negative result can also be very helpful. In this paper presents the negative results of a study performed but never published as part of our research investigating how code ownership impacts maintainability [8]. That paper was motivated by works of others [3, 4, 21] revealing an increased bug-prediction capability of models with some form of code ownership included as a predictor. We could confirm that clear ownership has a positive effect on code maintainability (measured by the combination of well-known source code metrics) as well. This result showed that common code is more likely to erode than code with clear ownership. This complements and generalizes bug-prediction studies on this topic. As part of this work, we wanted to show that the focus of developers also has a significant impact on software maintainability. This assumption was inspired by Di Nucci et al. [5] who investigated the impact of developer focus on post-release bugs. They found that a bug prediction model including the structural and semantic scattering metrics, which measure how close the modified code parts are structurally and conceptually overperforms the models not using these indicators.

We defined developer focus based on their definition of structural scattering [5], which is the distance between the files the developers work with at a time: the number of steps needed to take from the package of one file to another. The closest ones – having distance 0 – are those located in the same package. The

focus of a set of source files is the normalized aggregation of pairwise distances. The study of developer focus that is developer oriented (i.e., investigates the effect of what else the developer modified) would complement the code ownership study [8], which took a source code oriented approach (i.e., studies the effect of who else modified the same source code).

Formally, we investigated the following research questions:

RQ1: *Do more focused commits (i.e., commits affecting a set of source files with low scattering value) have better impact on maintainability compared to those of less focused commits?*

RQ2: *Do developers who were more focused in the past (i.e., the scattering value of the set of files they changed in the past 3 months are low) tend to commit more maintainable code, compared to less focused ones?*

To answer these questions we studied the code change history of three open-source systems and an industrial one. Our null-hypothesis was that there is no connection between developer focus and maintainability of the source code. Based on the statistical tests, unfortunately, we could not reject the null hypothesis, therefore we could not report evidence that developer focus impacts code maintainability.

All data for replicating our study is available as an online appendix at: `http://www.inf.u-szeged.hu/~ferenc/papers/DeveloperFocus/`

The remaining of the paper is organized as follows. Section 2 provides a brief overview of works that are related to this research. In Section 3 we present how we collected the data and what kinds of tests we performed. In Section 4 we present the results of the statistical tests. We conclude the paper in Section 5.

2 Related Work

Code ownership, which is very close to the topic of this paper, is widely investigated. The results are very contradictory: some researcher find significant correlation between code ownership and code quality, which others do not.

Nordberg and Martin [18] describe in their study four types of code ownership: product specialist, subsystem ownership, chief architect and collective ownership. They discuss the advantages and disadvantages of each model.

LaToza et al. performed two surveys and eleven interviews, conducted by software developers at Microsoft, regarding software development questions, and presented the results in article [17]. Some of the questions were related to code ownership, strongly related to this study. The authors formed an interesting statement: the code ownership can also be wrong, as if a code is understood and maintained by a single developer, it makes individuals too indispensable. As an alternative of individual code ownership, they investigated the team code ownership. This topic would be interesting also for us: to somehow determine teams and define team level code ownership and team level focus instead of individual level ones.

In their work Fritz et al. [13] investigated the frequency and the elapsed time of interactions on the code by developers. They asked questions to find

out if the developers can recall details about the source code: types of variables, types of parameters, method names, another method calls and methods which calls a specified method. The results supported their assumed hypothesis, the developers know their code that was modified by him/her frequently and recently better compared to foreign code. This study is strongly related to code ownership and developer focus, as a more focused developer might better recall source code elements regarding to the related code, compared to less focused developers.

Weyuker et al. [21] tried to enhance their defect prediction model by including the number of developers. They found that the achieved improvement is negligible, which is similar to the results we present in this study.

The same authors (Bell et al. [3]) tried to improve their defect prediction model considering the individual developers: they investigated whether files in a large system that are modified by an individual developer consistently contain either more or fewer faults than the average of all files in the system. They found negligible improvement: the study indicates that adding information to a model about which particular developer modified a file is not likely to improve defect predictions.

Hattori et al. [16] analyzed the problem of code ownership, especially finding the hidden co-authors. They considered in their model the developer interaction information as well. This could lead to a finer result also in case of developer focus.

The study by Bird et al. [4] investigated the effects of ownership on software quality. Under term of software quality they considered pre-release faults and post-release failures. They performed the analysis on binary and release level of the source code of Windows Vista and Windows 7. For a binary they defined the terms minor contributor (developers who contributed at most 5% of the total commits), major contributor (above 5%) and ownership (proportion of the commits of the highest contributor). Among others, they found that software components with many minor contributors had more failures than other software components. Moreover, the high level of ownership resulted in less defects.

Rahman et al. [20] introduced a code ownership and experience based defect prediction model, but instead of just considering the modifications performed on source file itself, they introduced a fine-grained level by analyzing the contributions to code fragments. This approach could be a good direction for future investigation of developer focus as well: besides considering source code package, other source code elements like class or function could also be taken into account.

Greiler et al. [14] defined several contributor-related metrics and used those for defect predicting model. Their findings confirm the original finding by Bird et al. [4] that code ownership correlates with code quality.

On the other hand, Foucault et al. [12] performed similar study as Bird et al. [4] on open-source systems, but they found that the relationship between ownership metrics and module faults is weak. The performed an in-depth analysis to find the reason of the different results of open-source and closed-source

software systems, and found that the reason is the distributions of contributions among developers and the presence of "heroes" in the open-source projects.

Di Nucci et al. [5] investigated the impact of developer focus on post-release bugs. They defined structural and semantic scattering (we implemented our developer focus value based on their structural scattering definition), and found that a bug predicting model including these metrics over performs models without these.

We also experienced these contradictory results. Our earlier model [8] considering the number of developers resulted in a not too strong but still significant result, but this study, considering developer focus did not show significant correlation at all.

3 Methodology

3.1 Overview

As we wanted to analyze the connection between developer focus and maintainability, we had to find a method to express these numerically. Neither of them are trivial concepts, and currently there are no exact definitions on how to compute them.

Considering maintainability, we performed the same calculation method that we applied in our previous studies [6–11]. We present the maintainability estimation method in detail in Section 3.3. The used quality model is capable of analyzing certain revisions of a system, therefore we chose to work on a per commit basis.

We calculated the developer focus values based on the definition of structural scattering by Di Nucci et al. [5]. We present the calculation details in Section 3.4. Section 3.5 describes the statistical tests we used to analyze the data. In Section 3.6, we explain our decisions made during the elaboration of the methodology.

3.2 Preliminary Steps

As first step we did some data cleaning. The analyzed software systems were all written in Java. As the used quality model considers Java source files only, we removed the non Java source files (e.g., xml files) from the input. If a commit contained non Java files only, then we also removed that one. So we worked on an input commit set that contained Java source files exclusively. Furthermore, each analyzed revision contained at least one affected Java file.

3.3 Estimation of the Maintainability Change

We used the ColumbusQM [1] probabilistic software quality model for estimating the maintainability value of every revision. It considers the following source code metrics: logical lines of code, the number of ancestors, the maximum nesting

level, the coupling between object classes, clone coverage, number of parameters, McCabe's cyclomatic complexity, number of incoming invocations, number of outgoing invocations, and number of coding rule violations. The basis of this model is the fact that there is a negative correlation between these metrics and software maintainability [15]. The quality mode compares these metrics of the analyzed system with those of other systems in a benchmark, and then aggretages the results of the comparisons by utilizing weights provided by developers.

From the study's viewpoint we treat this quality model as a black box. Details of this model is described the work of Bakota et al. [1]. The authors validated the model and they also revealed the correlation between the estimated quality value and the development costs [2]. The quality model results a real number between 0 and 1; better maintainability is indicated by higher value.

For each analyzed systems we calculated the maintainability values for every revision available in their version control systems. As next step we calculated the difference of the maintainability values of subsequent revisions, and then considered the sign of the result: positive, zero, or negative, indicating if the actual commit increased, did not considerably change or decreased the maintainability of the source code, respectively.

3.4 Calculation of Developer Focus

For calculating the focus of developers, we adopted the definition of *structural scattering*, described by Di Nucci et al. (see [5], page 243).

Let $CH_{d,p}$ be the the set of classes changed by a developer d during a time period p. The authors defined the *structural scattering* measure as:

$$StrScat_{d,p} = \frac{|CH_{d,p}|}{\binom{|CH_{d,p}|}{2}} \times \sum_{\forall c_i, c_j \in CH_{d,p}} [dist(c_i, c_j)]$$

where $dist$ is the number of steps to be taken in order to go from class c_i to class c_j. For example, the $dist$ between classes `pkg.entities.User` and `pkg.logic.util.Convert` is 3: `pkg.entities` $\rightarrow$ `pkg` $\rightarrow$ `pkg.logic` $\rightarrow$ `pkg.logic.util`. The multiplication factor at the beginning of the formula normalizes the distances between the code components and assigns a higher scattering to developers working on a higher number of code components in the given time period.

3.5 Comparison Tests

Once we had a maintainability change direction and a developer focus value for every commit in the revision history, we could check if there is any connection between the maintainability change direction and the focus of the developers. For this we performed several comparison tests, which fall into three categories: *commit-based*, *developer-based*, and *individual-based* tests. In all cases the basic setup of the tests was the same. First, we divided the commits into 3 subsets

based on the sign of maintainability changes, and analyzed their calculated developer focus values. How we defined the set of source code elements to which the developer focus should be calculated is described below. We omitted the neutral maintainability changes (i.e., no change in maintainability values happened), therefore we ended up with 2 sets of numbers:

- developer focus values of the commits with positive maintainability change, i.e. code quality increase, and
- developer focus values of the commits with negative maintainability change, i.e. code quality decrease.

The null hypothesis was that there is no significant difference between these values. The alternative hypothesis was that the developer focus values related to commits with positive maintainability changes are significantly lower than those related to negative maintainability changes, meaning that more focused commits (i.e., commits with low scattering value) are more likely to increase the maintainability.

We performed the Wilcoxon rank correlation test on the data, as this one is suitable for kind data we have (e.g., it is not normally distributed, or it has outliers). This test compares all the elements of the first data set with all the elements of the other one, taking all the possible combinations into consideration. According to our null hypothesis the number of "greater" elements should be roughly the same as the number of "less" elements. The alternative hypothesis expresses that the elements of one of the sets should be significantly higher than the elements of the other.

We used the R statistical program [19] for performing the tests, using the `wilcox.test()` function. As a result, we got p-values for all software systems we performed the test on. We ran tests with 3 different setups: *1)* for answering RQ1; *2)* and *3)* for answering RQ2:

Commit-based Comparison Tests In this case we used every commit for the developer focus calculation, and calculated the focus value considering all the files affected by that commit, as described in Section 3.4. As a result we got developer focus values for every commit.

Developer-based Comparison Tests In this case first we calculated a running developer focus value for every developer. For every commit we considered the developer who performed that commit and took all the files the developer changed in the previous 3 months. Therefore we simulated the process of forgetting. Furthermore, we omitted the commits containing more than 20 files, because that would have caused a big bias. For example, a directory rename could affect a large amount of source files, which would drastically increase the focus value of that developer for the next 3 months, but in the reality that developer has not lost the focus.

We defined the commit related focus value to be the focus value of the actual developer who performed the commit.

Individual-based Comparison Tests In this case we considered the version control history individually for every developer, considering commits performed only by that developer. This results in the same number of version control sub-histories as many developers contributed to the source code. We applied the methodology in every case as described in the developer-based comparison tests.

In order to avoid the non-explanatory results we excluded the developers who contributed too little, i.e. whose sub-history was too short to analyze. We considered only those developers who contributed at least 5 commits resulting a maintainability increase and at least 5 commits resulting a maintainability decrease.

3.6 Discussion

The *commit-based* comparison test is a rough approach, however our expectation was that if connection between maintainability change and developer focus existed, this could have been observed even by this method. In this case we did not consider earlier modifications of the developer, just the actual commit (i.e., we did not consider past focus).

On the other hand, *developer based* comparison tests are more fine-grained, and we thought of this as the main outcome of our study. The focus value calculation would be the same as in the commit-based case if we considered the union of all the contributions of the actual developer in the past 3 months (with the exception of huge commits). Our expectation was that considering this focus value the comparison tests would yield significant results.

In case of *individual-based* comparison tests we practically sliced the whole version control history to as many pieces as many developers contributed to that project, and kept only those that contained enough number of commits. This resulted in several tests per analyzed system (i.e., a separate test for every developer). With this test we wanted to ensure that we find per developer patterns even if we cannot find a general connection between developer focus and maintainability.

4 Results

4.1 Examined Software Systems

We executed the tests on four independent software systems. Our selection criteria for the subject systems were the following: availability of at least 1000 commits and at least 200% code increase during the analyzed period. We performed the analysis of the following 4 systems:

- **Ant** – a command line tool for building Java applications (`http://ant.apache.org`). All together 37 developers contributed at least once. The total number of available commits was more than 6000.
- **Gremon** – a proprietary greenhouse work-flow monitoring system (`http://www.gremonsystems.com`). 13 programmers took part in the development.

- **Struts 2** – a framework for creating enterprise-ready Java web applications (`http://struts.apache.org/`). The number of developers was 26.
- **Tomcat** – an implementation of the Java Servlet and Java Server Pages technologies `http://tomcat.apache.org`). 15 developers contributed at least once.

4.2 Results of the Statistical Tests

Table 1 contains the results of the commit-based and the developer-based comparison tests. The results are very sporadic, and the only significant p-value (0.003 in case of Gremon) seems to be rather casual than valid.

Similarly, the individual tests resulted sporadic results (each p-value is related to a particular developer but we ommited user names):

- **Ant**: 0.048, 0.082, 0.256, 0.373, 0.454, 0.485, 0.516, 0.576, 0.646, 0.673, 0.722, 0.762, 0.781, 0.830, 0.853, 0.883, 0.905, 0.933, 0.949, 0.991 (17 omitted)
- **Gremon**: 0.002, 0.038, 0.199, 0.211, 0.251, 0.410, 0.765, 0.787, 0.886, 0.940 (3 omitted)
- **Struts2**: 0.030, 0.100, 0.104, 0.144, 0.409, 0.434, 0.442, 0.600, 0.618, 0.651, 0.687, 0.774, 0.920, 0.950 (14 omitted)
- **Tomcat**: 0.020, 0.855, 0.872 (12 omitted)

Table 1. Resulting p-values of the Commit-based and Developer-based comparison tests (bold means significant p-value)

System	Commit	Developer
Ant	0.943	0.214
Gremon	0.208	**0.003**
Struts 2	0.571	0.687
Tomcat	0.921	0.951

4.3 Discussion

According to these results not just that we could not reject the null-hypothesis, but sporadic values did not even show us another possible direction with the research. For example, if the resulting p-values would have been close to 1, then we could perform the execution of the opposite tests, meaning that the more focused commits tend to result in maintainability erosion. Although it sounds counter-intuitive a possible explanation for this could be for example, that a more focused developer might be new on that project, and a lead developer's commits affect different parts of the code. But even that was not the case.

A bug in our software could also lead to negative results. To minimize the risk of this, we added unit tests to our implementation. Furthermore, we played around with different "forgetting" intervals (instead of the actually used 3 months) and huge commit threshold values (instead of the actually used 20), without relevant change in the results.

For omitting the too little contributions we performed the individual based comparison tests only for those who contributed at least 5 commits causing maintainability increase and same number for decrease. Using another value instead of 5 just changes the number of omitted values, but the actually calculated values will not change. We did not experience any relevant changes in the distribution of the resulting p-values by modifying this threshold (i.e., there is no such threshold value that would filter out only the high p-value results).

5 Conclusions and Future Work

As we already stressed in the introduction this is a publication of negative results. According to the statistic tests we performed we could not reject any of the null-hypotheses, either that more focused developers tend to contribute better quality code, or that more focused commits tend to improve code quality.

Out of the 8 main test (commit-based and developer-based tests for 4 systems) we performed, only one turned out to be significant: the developer-based comparison for the Gremon project, which is our only closed-source subject system. It would be worthwhile to investigate this in more detail because in an industrial environment a developer typically works on one singe project at a time, while in open-source environments a developer might contribute to several projects simultaneously. Bird et al. [4] and Greiler et al. [14] investigated some of their industrial products at Microsoft, and concluded that considering focus improve their fault prediction model, while Foucault et al. [12] could not achieve significant improvement by adding developer focus to an analysis of open-source systems. Do the open versus closed-source differences cause the contradictory results? This could be a topic of further investigations, however, it is hard to obtain the whole version control history, including the source code, for industrial software systems.

Acknowledgment

References

1. Bakota, T., Hegedűs, P., Körtvélyesi, P., Ferenc, R., Gyimóthy, T.: A probabilistic software quality model. In: 2011 27th IEEE International Conference on Software Maintenance. pp. 243–252. IEEE (2011)
2. Bakota, T., Hegedűs, P., Ladányi, G., Körtvélyesi, P., Ferenc, R., Gyimóthy, T.: A cost model based on software maintainability. In: 2012 28th IEEE International Conference on Software Maintenance (ICSM). pp. 316–325. IEEE (2012)
3. Bell, R.M., Ostrand, T.J., Weyuker, E.J.: The limited impact of individual developer data on software defect prediction. Empirical Software Engineering **18**(3), 478–505 (2013)
4. Bird, C., Nagappan, N., Murphy, B., Gall, H., Devanbu, P.: Don't touch my code!: examining the effects of ownership on software quality. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. pp. 4–14. ACM (2011)
5. Di Nucci, D., Palomba, F., Siravo, S., Bavota, G., Oliveto, R., De Lucia, A.: On the role of developer's scattered changes in bug prediction. In: Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on. pp. 241–250. IEEE (2015)
6. Faragó, C.: Variance of source code quality change caused by version control operations. Acta Cybernetica **22**, 35–56 (2015)
7. Faragó, C., Hegedűs, P., Ferenc, R.: The impact of version control operations on the quality change of the source code. In: Computational Science and Its Applications (ICCSA), pp. 353–369. Springer (2014)
8. Faragó, C., Hegedűs, P., Ferenc, R.: Code ownership: Impact on maintainability. In: Computational Science and Its Applications–ICCSA 2015, pp. 3–19. Springer (2015)
9. Faragó, C., Hegedűs, P., Ferenc, R.: Cumulative code churn: Impact on maintainability. In: 15th IEEE International Working Conference on Source Code Analysis and Manipulation–SCAM 2015 (2015)
10. Faragó, C., Hegedűs, P., Ladányi, G., Ferenc, R.: Impact of version history metrics on maintainability. In: Proceedings of the 8th International Conference on Advanced Software Engineering & Its Applications (ASEA). pp. 30–35. IEEE Computer Society (2015)
11. Faragó, C., Hegedűs, P., Végh, Á.Z., Ferenc, R.: Connection between version control operations and quality change of the source code. Acta Cybernetica **21**, 585–607 (2014)
12. Foucault, M., Falleri, J.R., Blanc, X.: Code ownership in open-source software. In: Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering. p. 39. ACM (2014)
13. Fritz, T., Murphy, G.C., Hill, E.: Does a programmer's activity indicate knowledge of code? In: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering. pp. 341–350. ACM (2007)
14. Greiler, M., Herzig, K., Czerwonka, J.: Code ownership and software quality: a replication study. In: Proceedings of the 12th Working Conference on Mining Software Repositories. pp. 2–12. IEEE Press (2015)
15. Gyimóthy, T., Ferenc, R., Siket, I.: Empirical validation of object-oriented metrics on open source software for fault prediction. Software Engineering, IEEE Transactions on **31**(10), 897–910 (2005)

16. Hattori, L., Lanza, M.: Mining the history of synchronous changes to refine code ownership. In: Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on. pp. 141–150. IEEE (2009)
17. LaToza, T.D., Venolia, G., DeLine, R.: Maintaining mental models: a study of developer work habits. In: Proceedings of the 28th international conference on Software engineering. pp. 492–501. ACM (2006)
18. Nordberg III, M.E.: Managing code ownership. Software, IEEE **20**(2), 26–33 (2003)
19. R Core Team: R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria (2015), `http://www.R-project.org/`
20. Rahman, F., Devanbu, P.: Ownership, experience and defects: a fine-grained study of authorship. In: Proceedings of the 33rd International Conference on Software Engineering. pp. 491–500. ACM (2011)
21. Weyuker, E.J., Ostrand, T.J., Bell, R.M.: Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models. Empirical Software Engineering **13**(5), 539–559 (2008)