Which Refactoring Reduces Bug Rate?

Idan Amit
idan.amit@mail.huji.ac.il
The Hebrew University of Jerusalem
Jerusalem, Israel

Dror G. Feitelson
feit@cs.huji.ac.il
The Hebrew University of Jerusalem
Jerusalem, Israel

ABSTRACT

We present a methodology to identify refactoring operations that reduce the bug rate in the code. The methodology is based on comparing the bug fixing rate in certain time windows before and after the refactoring. We analyzed 61,331 refactor commits from 1,531 large active GitHub projects. When comparing three-month windows, the bug rate is substantially reduced in 17% of the files of analyzed refactors, compared to 12% of the files in random commits. Within this group, implementing 'todo's provides the most benefits. Certain operations like reuse, upgrade, and using enum and namespaces are also especially beneficial.

CCS CONCEPTS

• **Software and its engineering → Software development techniques**.

KEYWORDS

Code quality, refactoring, machine learning

ACM Reference Format:
Idan Amit and Dror G. Feitelson. 2019. Which Refactoring Reduces Bug Rate?. In *The Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE'19), September 18, 2019, Recife, Brazil.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3345629.3345631

1 INTRODUCTION

Our goal is to find actionable recommendations for software quality improvement. Refactoring is an established approach to improving code quality and promoting maintainability and continued development [6]. Many different refactoring operations have been proposed. But which ones provide the best benefits?

Refactors are change operations intended to improve code quality without changing functionality. We use a methodology that analyzes changes over time. The metric we use is bug fixing rate: Out of all the commits in a given period, what fraction are bug fixes. We identify corrective and refactor commits using linguistic models applied to the commit message. Once we identify the refactors, we compare the bug-fix rate as reflected by corrective commits before and after the refactor.

PROMISE'19, September 18, 2019, Recife, Brazil

ACM ISBN 978-1-4503-7233-6/19/09…$15.00
https://doi.org/10.1145/3345629.3345631

Note that even when we see an improvement around a change, it still doesn't imply causality, and we cannot be sure that the operation is recommended. The change might be accidental, or due to other reasons. However, this methodology enables us to explore, identify candidates, and follow up with focused experiments.

Refactors are 12% of the commits in large active GitHub projects, but not all of them are equally suitable for analysis. Some don't have enough other commits around them, some involve a huge number of files, and some are not done on identified source code. In order to explore and identify effective refactors, we built a dataset of 'clean' refactors—commits of one non-test file with enough context—and their messages. These commits were labeled as positive if they reduced the bug-fix rate substantially. This reduces the problem to supervised learning, where we can explore the attributes that characterize commits that reduce the bug rate.

1.1 Related Work

Refactoring, first suggested by Opdyke [15], is "improving the design of existing code" [6]. Prior work investigated the influence of refactoring on quality [1, 3, 4, 8, 14, 17, 19]. In general, this research showed mixed results of positive and negative influence, of small size. While prior work asks "Does refactoring have a positive influence?", we ask "Which refactor operations have a positive influence?", enabling us to identify such operations.

To check the influence of refactoring one has to identify refactoring. Most work focused on the refactor technique. Tsantalis et al. [18] developed RMiner for Java which has very high performance: 98% precision and 87% recall. However, they built an Abstract Syntax Tree of the code in order to identify changes. We did the analysis in Google's BigQuery GitHub schema where the code per commit is not available. Other approaches are manual labeling, which is limited in scope, or Ref-Finder [9] with precision of only 35% [16]. We identify refactors and corrective commits using linguistic analysis of commit messages, an idea that is commonly used for defect prediction [7].

1.2 Our Contribution

We present a way to find bug-rate reducing operations. We provide scalable (analyzing millions of commits) accurate models to identify corrective and refactor commits, based only on their messages. We are agnostic to the refactor technique used and therefore can explore and identify new efficient operations. We provide empirical evidence for code rot [5, 10], and show that refactoring slows the decay rate. This is a demonstration of Lehman's 7th law of software evolution, which states that software quality will appear to decline unless it is rigorously maintained [11]. Our conclusions recommend on refactor operations that improve quality.

2 CHANGE IDENTIFICATION AND SCOPE

2.1 Linguistic Models for Commit Types

There are 10,845 open source projects in GitHub with 500+ commits in 2018. We analyzed the 1,531 non-fork projects [2], since forks are very close to their origin, are redundant, and distort the distribution.

We sampled uniformly random commits from the large active projects. We labeled the commits following the taxonomy of Lientz et al. [13] into corrective, adaptive, and perfective. We also labeled refactor as a sub type of perfective. We then built a linguistic model for commit messages, identifying positive occurrences and removing invalid ones (e.g., due to negative context, modals, or being part of a different unrelated term). We used a test set of 1,100 samples on which we evaluated the classifiers. The corrective model has accuracy of 89%, precision of 84%, and recall of 69%. For the refactor model, the accuracy is 93%, precision is 80%, and recall is 61%. For our refactor use-case we need high precision. Moreover, knowing the precision value enables us to estimate the real influence, considering that 20% of the hits are not refactors. Our analysis doesn't depend directly on the recall, yet the higher the recall, the more cases we will have. See supplementary materials for more details.

2.2 Building a Refactors Dataset

Using the models we identified 749,603 commits as refactors. The commits differ by their suitability to change analysis. A refactor might be subject to noise due to few commits in context and application to many files. We look for clean, less noisy, commits. Yet, the more restrictive the definition of a clean refactor, the fewer cases we have to analyze. Hence, we are in a tradeoff between quantity and cleanness.

The following describes the 176,309 refactors done during 2018, on which we based our decisions regarding the scope of refactors to analyze and their context. The full distributions and generating code are part of the supplementary materials.

First, we are interested only in refactoring done on source code. We used the 28 major source code filename extensions, which cover 49% of GitHub files. Out of all the (file, refactor commit) pairs, 57% belong to a major source code extension.

Another set of files that we would like to scope out are test files, since the goal and behavior of tests is different from regular code. Moreover, a refactor involving a test is likely to refactor the tested file and not the test file. Since the code is not available in BigQuery, we identify test files by matching their path with the pattern 'test'[12, 20], which matched 21%.

We define the context of a refactor as the commits made to the same file before and after the refactor. In order to identify a change, we need at least one commit before and after the refactor, and more are needed for a robust analysis. In a context that spans three months, 57% had fewer than 10 commits before the refactor and 56% had fewer than 10 commits after it. In a context that spans six months the figures were 56% and 48%. Aiming for clean results we examined the 61,331 refactor commits with at least 10 commits in 3 months on either side, and involving a single non-test file.

3 REFACTOR INFLUENCE ANALYSIS

We first explain the methodology and then present results.

3.1 Linguistic Identification of Useful Refactors

Refactors are diverse in type, size, goal, subject, and implementation. As we show below, their effect on bug rate has a high standard deviation. This hints that there might be different types of refactors with different effects.

In order to identify the most effective refactors, we retrieved the commit messages from our refactor dataset and tokenized them. We labeled them by whether the refactor reduced the bug rate by at least 0.1. This enabled us to use machine learning to predict improvement and evaluate influence. However, the number of tokens is very high, the same semantic meaning might be represented by different tokens, and all probabilities are subject to noise. Therefore, we used linguistic and domain knowledge to map the tokens into groups, e.g. considering 'reorganizes', 'reorganized' and 'repackaging' as members of the 'reorganize' group.

3.2 Using Coupling as a Metric

Linguistic analysis as described above is based on what the developer intended to do in the refactor, and documented in the commit message. But the refactor might fail to fulfill the intent or have other side effects. It is therefore interesting to also look at the effect of software properties that just happened to change. We use coupling, which is a metric of software quality, and examine the influence of a reduction in the coupling on the bug-fix rate.

Zimmermann et al. [21] showed that co-changes analysis, namely files that change in the same commit, can be used to detect coupling. We use the idea as a file-level metric for coupling based on the size of co-changes. A commit is a unit of work ideally reflecting the completion of a task. It should contain only the files relevant to that task. A large number of files needed for a task means coupling. Therefore, the metric is the average number of files in a commit.

The same approach can be applied to other software metrics such as length, readability, and complexity.

3.3 Influence of Refactoring

In order to analyze the influence of refactors, we should first know what happens without them. We compared the bug-fix rates of files in two adjacent three-month windows and saw that only 42% of the files had a lower bug-fix rate in the later window. The average difference in the bug-fix rate is only 0.004, with standard deviation of 0.2. Probability of improving by 0.1 or more was 12% (with 10 commits, 0.1 means one commit). These results indicate code rot, yet show that code quality decreases slowly and with a variance that is much larger than the change.

We present the influence of refactors in table 1. We examined clean refactors with at least 10 commits in a three-month window before them and in two such windows after them. The requirement for the second window reduces the dataset by about 35%.

The average change in the bug rate is small for all refactor types, and in all but one it is positive (an increase in bug rate). However, the standard deviation is large, so there are many cases where the refactoring does help. The results are sorted according to the probability of a substantial reduction in the bug rate (emphasized), where "substantial" is taken to be at least 0.1 in a 3-month context. In all

Table 1: Effect of refactor operations on bug rate

Metric:		difference	probability to improve			
		Avg±stddev	by >0.1	by >0	by >0	by >0
Window before:		3mon	3mon	3mon	3mon	6mon
Window after:		3mon	3mon	3mon	4-6mn	6mon
Action	Commits					
todo	256	-0.003±0.137	**0.248**	0.493	0.523	0.500
feedback	149	0.005±0.137	**0.228**	0.479	0.407	0.419
reuse	122	0.027±0.136	**0.211**	0.398	0.451	0.511
upgrade	79	0.002±0.134	**0.205**	0.446	0.566	0.506
SE goals	995	0.003±0.137	**0.205**	0.485	0.527	0.506
sp/tab	358	0.000±0.132	**0.202**	0.486	0.469	0.499
improve	3,193	0.005±0.135	**0.196**	0.482	0.484	0.486
optimization	1,141	0.005±0.135	**0.196**	0.465	0.475	0.457
refactor	4,241	0.006±0.133	**0.186**	0.465	0.505	0.489
enum/ns	261	0.004±0.121	**0.184**	0.494	0.494	0.523
unneeded	7,167	0.001±0.129	**0.180**	0.501	0.496	0.502
rework	850	0.013±0.133	**0.178**	0.449	0.466	0.401
baseline	**39,944**	**0.005±0.127**	**0.172**	**0.475**	**0.486**	**0.476**
reorganize	423	0.013±0.144	**0.168**	0.410	0.487	0.462
rename	2,492	0.003±0.123	**0.168**	0.463	0.481	0.459
simplify	2,605	0.004±0.122	**0.164**	0.473	0.471	0.461
clean	595	0.009±0.134	**0.162**	0.460	0.482	0.482
coupling	10,180	0.028±0.133	**0.135**	0.403	0.447	0.459
style	196	0.036±0.137	**0.132**	0.397	0.470	0.420
spelling	47	0.046±0.128	**0.080**	0.380	0.500	0.360

cases but one this was larger than the 12% found for general commits. The next column shows the probability for *any* improvement, and the last two consider alternative time windows. "Commits" is the number of refactors we analyze. Note that some of the results are based on a small number of cases.

The rows of the table represent the refactoring operations. The baseline row (emphasized) presents the influence of a general refactor. Operations that may require explanation are 'feedback' - with external feedback (human as code review or mechanical as pylint), 'enum/ns' - software engineering constructs like enum or namespace, 'SE goals' - explicit reference to a software goal like abstraction, 'unneeded' - removing unneeded code, 'sp/tab' - the whitespace/tabs wars common in the Python community, 'refactor' - explicitly mentioning the term. 'Coupling' is refactors that had a side-effect of reducing the number of non-test files in the commit by at least one file. The rest of the operations are a textual evidence of the name of the operation and its semantically equivalent terms. A refactor can involve several operations.

Only one operation had a probability for improvement of 50% in the next three months. However, in 75% of them the probability of improvement was better than the 42% expected for general commits. Observing the difference average and standard deviation it is clear that all changes are small, usually reducing the quality and with a high variance. This explains the mixed results in prior work. None of the operations is a silver bullet or even reaches 60% probability of improvement and a substantial reduction in bug rate.

Another result is that four to six months after the refactor many operations are better than after one to three months. This might suggest that a refactor disrupts the system and might cause more bugs in the short term. But good refactors have a return on investment in the long term, and for some operations the six-month bug rate shows improvement.

3.4 Possible Explanations

Once we can identify influential operations, the first question that comes to mind is "Why does it influence in this way?" We don't claim to provide answers here but to suggest ideas that should be investigated on their own.

The probability of reducing the bug rate by 0.1 between two 3-month periods is 12%. Doing a refactor raises the probability to 17%. The leading operation is 'todo', though based on a small number of cases. It is interesting to further investigate why it is so influential. A possible reason is that 'todo's actually reflect what the developers themselves think should be done, based on their knowledge of the code, and it is indeed advisable to act on this input.

'Feedback' and 'SE goals' indicate a change that is supported by an external influence and guided. Maybe the identification of a proper target contributes to the success. 'Feedback' is not a specific change but a result of consulting, and its substantial improvement probability is 32% higher than that of a baseline refactor.

'Reuse' and 'upgrade' should have been a free lunch, using an already-existing component. However, the change itself led to short term bugs and returning the investment only in the longer term.

The influence of 'improve' is lower, suggesting that we might not be as good as we think in identifying needed improvements. This is even more so with 'clean' and 'reorganize', which are slightly less beneficial than a general refactor, and 'rework', which is perhaps marginally better.

Simplification is one of the most advocated principles in software engineering and in general. One would expect that the influence of simplification would be high, but our results indicate it is lower than a general refactor. It will be interesting to investigate if simplification refactors indeed improve complexity metrics.

Interestingly, other than 'rename' and 'unneeded', we didn't identify the linguistic expression of any refactor technique (e.g., 'extract method', 'pull up/down'). This might suggest that the context and goal, and not the technique, are the cause of influence. A focused study is needed to verify this.

Reduced coupling is slightly worse than no refactor in the first 3 months, and improves later. This might hint that a large reduction in coupling is somewhat positive but involves destabilizing the system.

3.5 Influence on the System and Influence of the Developer

The linguistic exploration led to some surprising results. Consider the removal of unneeded code, whitespace wars, style, rename, and spelling. The compiler is indifferent to all of them, yet we see that they have an effect. Hence, they influence via the developer, who might make more bugs, e.g. due to confusing variable names.

The terms of whitespace wars, style, and spelling were not part of our definition of a refactor. But their influence was big enough to be observed in our refactor model hits. It is also possible that developers who pay attention to style, pay attention to quality in

general. In order to investigate their influence regardless of our refactor model, one should develop a model for them and consider all their occurrences.

4 THREATS TO VALIDITY

We checked differences in bug-fix rates before and after a refactor as a measure of quality improvement. The time windows used might be too long or too short. Since our context is rather long, it is possible that different causes interfere with the change, a threat we try to control by using a large number of cases.

The use of bug fixes might be misleading in case the refactor does not lead to more bugs but helps to find bugs more efficiently. For example, simplifying the code might reveal old bugs and increase the future bug-fix rate.

We don't know how a developer decides to do a refactor. However, it is not uniformly selecting a random file each day. The refactors done depend on the history and goals, and this might bias our data. If one decides to refactor a file after a period that had many fixes by chance, the regression towards the mean might be seen as an improvement. External causes, like a quality assurance blitz, might also change the probability of finding and fixing a bug.

In order to analyze the refactors we needed a suitable context. The refactors in the scope, and even more the clean refactors, are a small part of all the refactors. A refactor done on an extensively changing file might not represent other refactors. Finding recommendable operations using clean commits and verifying them on all commits will enable validation of the results.

The linguistic models for corrective and refactor commits were built and estimated using labeled data sets. The models reach high precision by not only identifying a term occurrence but also noticing that "error message" and "this is a feature and not a bug" do not indicate a corrective commit. While refactor operations are part of the model, we don't have a labeled data set and performance evaluation for them. A text occurrence is usually positive, but validation is needed.

5 FUTURE WORK AND CONCLUSIONS

We presented a method to identify refactor operations and evaluate their effectiveness in reducing the bug rate. We provide results on known refactoring operations and identify new ones.

In general, refactors are instrumental in reducing bug rates due to code rot. A recommendation depends on the metric of interest. Almost all refactor operations have a better probability for a substantial improvement or improvement after four to six months than a general commit. Being conservative, we recommend operations with at least 50% for improvement in the next six months: do your 'todo's, remove unneeded code, aim to improve software goals, upgrade and reuse, and use enums and namespaces.

Many other metrics can be used as both the changed metric in the refactor or the target metric we value. Some of them have immediate influence (e.g., a refactor that shortens a file length), making the change cleaner and the analysis results more robust. Many other aspects might influence the effectiveness of refactoring: developer's familiarity with the code, experience, file age, etc. The high variance of influence gives hope to finding more effective operations. Using this method, and further developing the method itself, can lead to more actionable recommendations.

Supplementary Materials

See https://github.com/evidencebp/Which-Refactoring-Reduces-Bug-Rate

Acknowledgements

This research was supported by the ISRAEL SCIENCE FOUNDATION (grant No. 832/18).

REFERENCES

[1] M. Alshayeb. Empirical investigation of refactoring effect on software quality. *Information and Software Technology*, 51(9):1319–1326, 2009.
[2] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu. The promises and perils of mining git. In *6th IEEE International Working Conference on Mining Software Repositories*, pages 1–10, May 2009.
[3] B. D. Bois and T. Mens. Describing the impact of refactoring on internal program quality. In *Proceedings of the International Workshop on Evolution of Large-scale Industrial Software Applications*, pages 37–38, 2003.
[4] O. Chaparro, G. Bavota, A. Marcus, and M. D. Penta. On the impact of refactoring operations on code quality metrics. In *IEEE International Conference on Software Maintenance and Evolution*, pages 456–460, Sep. 2014.
[5] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does code decay? Assessing the evidence from change management data. *IEEE Trans. Softw. Eng.*, 27(1):1–12, Jan 2001.
[6] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Signature Series (Fowler). Pearson Education, 2018.
[7] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Trans. Softw. Eng.*, 38(6):1276–1304, Nov 2012.
[8] S. H. Kannangara and W. M. J. I. Wijayanayake. Impact of refactoring on external code quality improvement: An empirical evaluation. In *Intl. Conf. Advances in ICT for Emerging Regions (ICTer)*, pages 60–67, Dec 2013.
[9] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit. Ref-finder: a refactoring reconstruction tool based on logic query templates. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 371–372, 2010.
[10] M. M. Lehman. On understanding laws, evolution, and conservation in the large-program life cycle. *J. Syst. & Softw.*, 1:213–221, 1980.
[11] M. M. Lehman. Laws of software evolution revisited. In *European Workshop on Software Process Technology*, number 5, pages 108–124. Springer Verlag, Oct 1996. Lect. Notes Comput. Sci. vol. 1149.
[12] S. Levin and A. Yehudai. The co-evolution of test maintenance and code maintenance through the lens of fine-grained semantic changes. In *IEEE International Conference on Software Maintenance and Evolution, (ICSME)*, pages 35–46, 2017.
[13] B. P. Lientz, E. B. Swanson, and G. E. Tompkins. Characteristics of application software maintenance. *Comm. ACM*, 21(6):466–471, Jun 1978.
[14] R. Moser, P. Abrahamsson, W. Pedrycz, A. Sillitti, and G. Succi. A case study on the impact of refactoring on quality and productivity in an agile team. In B. Meyer, J. R. Nawrocki, and B. Walter, editors, *Balancing Agility and Formalism in Software Engineering*, pages 252–266. Springer Berlin Heidelberg, 2008.
[15] W. F. Opdyke. Refactoring object-oriented frameworks. Technical report, 1992.
[16] G. Soares, R. Gheyi, E. Murphy-Hill, and B. Johnson. Comparing approaches to analyze refactoring activity on software repositories. *Journal of Systems and Software*, 86:1006–1022, 04 2013.
[17] K. Stroggylos and D. Spinellis. Refactoring–does it improve software quality? In *Proceedings of the 5th International Workshop on Software Quality*, WoSQ '07, pages 10–, Washington, DC, USA, 2007. IEEE Computer Society.
[18] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 483–494, 2018.
[19] D. Wilking, U. F. Kahn, and S. Kowalewski. An empirical evaluation of refactoring. *e-Informatica*, 1(1):27–42, 2007.
[20] A. Zaidman, B. Van Rompaey, A. van Deursen, and S. Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, 16(3):325–364, Jun 2011.
[21] T. Zimmermann, S. Diehl, and A. Zeller. How history justifies system architecture (or not). In *6th Intl. Workshop Principles of Software Evolution*, Sept. 2003.