The Buggy Side of Code Refactoring: Understanding the Relationship between Refactorings and Bugs

Isabella Ferreira[1], Eduardo Fernandes[1], Diego Cedrim[1], Anderson Uchôa[1], Ana Carla Bibiano[1], Alessandro Garcia[1], João Lucas Correia[2], Filipe Santos[2], Gabriel Nunes[2], Caio Barbosa[2], Baldoino Fonseca[2], Rafael de Mello[1]
{iferreira,emfernandes,dcgrego,auchoa,abibiano,afgarcia,rmaiani}@inf.puc-rio.br
{jlmc,filipebatista,gabrielnunes,cbvs,baldoino}@ic.ufal.br
[1]Pontifical Catholic University of Rio de Janeiro, Brazil
[2]Federal University of Alagoas, Brazil

ABSTRACT

Code refactoring is widely practiced by software developers. There is an explicit assumption that code refactoring improves the structural quality of a software project, thereby also reducing its bug proneness. However, refactoring is often applied with different purposes in practice. Depending on the complexity of certain refactorings, developers might unconsciously make the source code more susceptible to have bugs. In this paper, we present a longitudinal study of 5 Java open source projects, where 20,689 refactorings, and 1,033 bug reports were analyzed. We found that many bugs are introduced in the refactored code as soon as the first immediate change is made on it. Furthermore, code elements affected by refactorings performed in conjunction with other changes are more prone to have bugs than those affected by pure refactorings.

CCS CONCEPTS

• **Social and professional topics** → **Software maintenance**; • **Software and its engineering** → **Software defect analysis**; • **General and reference** → *Empirical studies*;

KEYWORDS

Refactoring, bug proneness, software maintenance, empirical study

ACM Reference format:
Isabella Ferreira[1], Eduardo Fernandes[1], Diego Cedrim[1], Anderson Uchôa[1], Ana Carla Bibiano[1], Alessandro Garcia[1], João Lucas Correia[2], Filipe Santos[2], Gabriel Nunes[2], Caio Barbosa[2], Baldoino Fonseca[2], Rafael de Mello[1] . 2018. The Buggy Side of Code Refactoring: Understanding the Relationship between Refactorings and Bugs. In *Proceedings of 40th International Conference on Software Engineering, Gothenburg, Sweden, May 2018 (ICSE'18),* 2 pages.
https://doi.org/10.475/123_4

1 INTRODUCTION

Code refactoring is a program transformation used to improve the structure of a program while preserving its observable behavior [3]. Refactoring, in practice, consists of two significantly different tactics, namely root-canal refactoring and floss refactoring [4]. Developers apply *root-canal refactoring* when they only perform refactoring operations in a change. Developers apply *floss refactoring* when they perform refactoring operations together with other program modifications in a single change.

One could assume that developers are more often intended at improving structural quality through root-canal refactoring. However, in both cases, one could also expect that structural quality of a program is somehow improved through refactoring and, therefore, also making the code less prone to bugs in the future. In fact, there is always an explicit assumption that code refactoring improves the structural quality of a program, thereby reducing the likelihood of future bugs on the refactored program elements [5]. However, this assumption might not always hold. Developers might unconsciously make the source code more susceptible to have bugs depending on the complexity of the refactoring operation.

Unfortunately, there is limited understanding about the relationship between refactorings and bugs, nor the influence of root-canal and floss refactorings on bugs. Instead, existing studies [6, 8] merely tend to focus on confirming the intuition that the higher the density of refactorings in a program, the lower the number of bugs in the affected code elements. However, a study controversially suggests that certain types of refactorings may induce bugs [1]. To better understand the relationship between refactorings and bugs, we performed a longitudinal study of 5 Java open source software projects, which consist of a tally of where 20,689 refactorings, and 1,033 bug reports. We analyze to what extent refactored code elements are susceptible to have bugs, i.e., if bugs tend to emerge soon in refactored code. This analysis is supported by verifying the *distance* in number of changes between the commit, where the refactoring was performed, and the commit where the bug emerged in the refactored code element. We also analyze the relationship between refactoring tactics and bugs.

2 STUDY DESIGN AND RESULTS

Figure 1 illustrates the study phases designed for investigating the relationship between refactorings and bugs. In this study, we aim to answer the following question: *What is the bug proneness of refactored code elements?*. To do that, we assess how distant (Section 1) a bug appears in the software project after a refactoring operation takes place. A detailed description of each study phase and all study artifacts are available at the study website [2].

ICSE'18, May 2018, Gothenburg, Sweden

ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

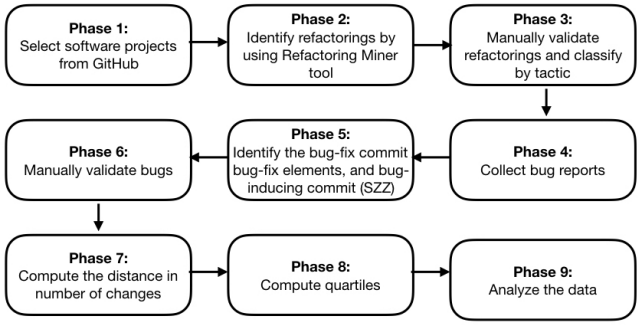

Figure 1: Study Phases

Table 1 presents the distances results. We computed the quartiles for the distances values and classified them into distance categories. The minimum value of distance represents that a bug is *Too Close* of the refactoring code commit. The first quartile (25%) means that the bug appeared in a *Very Close* distance of the refactored code commit. Similarly, the median (50%) of the project's data is classified as *Close*. The third quartile (75%) means that the bug is *Distant*, and the maximum value of distance means that the bug is *Very Distant* of the refactored code commit.

Table 1: Bug proneness of refactored code elements

Project	N	Mean	Stdev	Too Close	Very Close	Close	Distant	Very Distant
Ant	7	41.86	12.79	28	31	41	51	60
Derby	23	35.65	32.40	1	11	20	59	104
Okhttp	52	35.12	26.93	1	6	36	56	95
Presto	55	27.25	27.15	1	8	16	45	20
Tomcat	11	13.64	15.62	1	2	5	14	37

By looking at the table, we can see that, except for the Ant project, in 25% of the project data (*Very Close* column) is necessary up to 11 changes to emerge a bug in the refactored code element. Similarly, in 50% of project data (*Close* column), it is necessary up to 36 changes so that the bug emerges after the refactoring operation. However, we have found many cases of bugs that were too close (1 change) away from refactorings in most projects (see column 5). This result is interesting because it suggests that certain cases of refactoring operations may have a direct effect on the bug proneness of a code element. Thus, in contrast to the literature (Section 1), we observed that refactoring operations might lead to the occurrence of bugs. In fact, Śliwerski et al. [7] found that refactoring operations usually do not induce bugs. The different study settings may have affected the results. For instance, we systematically identify refactoring operations based on both tool support and manual validation, while previous work relies on change analysis, without methodically validating whether a change represents or not a refactoring operation. We have also analyzed a different set of projects, and we have considered 5 software projects against only 2 of previous work [7]. In summary, we answer our RQ by observing that refactored code elements are associated with the further emergence of bugs.

> ***Finding 1.*** Many bugs are introduced in the refactored code as soon as the first immediate changes are made on it.

Analysis per Refactoring Tactic. In the previous analysis, we assessed the bug proneness of refactored code regardless the tactic of developers when refactoring. However, as discussed in Section 1, developers often use two refactoring tactics, namely root-canal and floss refactoring. Thus, aimed at understanding whether different tactics have different effects on the bug proneness of refactored code, we assess the distance values between refactorings and bugs per refactoring tactic. To do that, we manually analyzed a randomly selected sample of 2,119 refactorings in five software projects. To support our analysis, Table 2 presents the distance results per refactoring tactic.

Table 2: Bug proneness of refactored elements per refactoring tactic

Project	Very Close		Close		Distant		Very Distant	
	Root	Floss	Root	Floss	Root	Floss	Root	Floss
Ant	60	30	60	33	60	39	60	46
Derby	30	5	34	8	47	20	61	104
Okhttp	10	2	18	3	24	5	53	12
Presto	1	1	4	5	14	25	32	105
Tomcat	1	1	1	2	1	5	1	9

Our results suggest that 50% of the distance values vary in a range from 1 to 60 for root-canal refactoring, and from 1 to 33 for floss refactoring. This implies that, when developers are refactoring probably with the intention of exclusively improving the code structural quality (root-canal refactoring), bugs appear farther from the refactored code commit if compared to floss refactoring. Instead, when developers have other primary concerns rather than improving software quality (floss refactoring), the bugs tend to appear much closer to the refactored code commit.

> ***Finding 2.*** Code elements affected by floss refactoring are closer to bugs than code elements affected by root-canal refactoring.

3 ACKNOWLEDGMENT

This work is funded by CAPES/Procad (grant # 175956), CNPq (grants # 309884/2012-8, 483425/2013-3 and 477943/2013-6), FAPERJ (E26-102.166/2013) and FAPERJ (grant # 102166/2013 and 22520 7/2016).

REFERENCES

[1] Gabriele Bavota et al. 2012. When does a refactoring induce bugs?. In *SCAM*.
[2] Isabella Ferreira et al. 2018. Research website. (2018). https://isabellavieira57.github.io/icse2018/index.html
[3] Martin Fowler. 1999. *Refactoring*. Addison-Wesley Professional.
[4] Emerson Murphy-Hill, Chris Parnin, and Andrew P Black. 2012. How we refactor, and how we know it. *TSE* 38, 1 (2012).
[5] F. Palomba et al. 2016. Smells Like Teen Spirit: Improving Bug Prediction Performance Using the Intensity of Code Smells. In *ICSME*.
[6] Jacek Ratzinger, Thomas Sigmund, and Harald C Gall. 2008. On the relation of refactorings and software defect prediction. In *MSR*.
[7] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes?. In *ACM SIGSOFT Software Engineering Notes (SEN)*. 1–5.
[8] Peter Weißgerber and Stephan Diehl. 2006. Are refactorings less error-prone than other changes?. In *MSR*.