A Preliminary Investigation of Self-Admitted Refactorings in Open Source Software

Di Zhang, Bing Li, Zengyang Li*, Peng Liang
School of Computer Science, Wuhan University
Wuhan, China

***Abstract*—In software development, developers commit code changes to the version control system. In a commit message, the committer may explicitly claim that the commit is a refactoring with the intention of code quality improvement. We defined such a commit as a self-admitted refactoring (SAR). Currently, there is little knowledge about the SAR phenomenon, and the impact of SARs on software projects is not clear. In this work, we performed a preliminary investigation on SARs with an emphasis on their impact on code quality using the assessment of code smells. We used two non-trivial open source software projects as cases and employed the PMD tool to detect code smells. The study results shows that: (1) SARs tend to improve code quality, though a small proportion of SARs introduced new code smells; and (2) projects that contain SARs have different results on frequently affected code smells.**

Keywords-self-admitted refactoring; code smell; case study; code quality

I. INTRODUCTION

Software systems are evolving over time once they are delivered. Software evolution often comprises up to 75% of the costs of software development [1]. However, the decrease in quality and increase of complexity push developers and practitioners to come up with flexible, maintainable, and extensible techniques for improving software quality and reducing change costs. One of these techniques is refactoring that is "*the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure* [2]." Software maintainability can be well indicated by code smells [3], while refactoring is a recommended daily practice and considered as an effective way to fix code smells [1].

Interestingly, developers often explicitly claim, in the commit messages of version control systems, that their modifications to the software system are refactorings. It means, by definition, that the maintainability of the software system is expected to have been improved. We call such code modifications, claimed as refactorings by developers, self-admitted refactorings (SARs). The concept of SAR is inspired by [4, 5], in which the term of self-admitted technical debt was introduced. Commits of a project can be divided into two categories: SARs and non-SARs (commits without SARs). Currently, there is little empirical evidence on whether SARs do improve the structure quality of the code.

*Corresponding author. E-mail: zengyangli@whu.edu.cn.
DOI reference number: 10.18293/SEKE2018-081

II. BACKGROUND AND RELATED WORK

A. Code Smells and Refactoring

Fowler *et al.* proposed to use code smells to indicate the structure quality issues in code, which are possible refactoring opportunities [2]. They defined 22 common code smells, e.g., duplicated code [2]. Later, some new code smells were proposed. For instance, Kerievsky proposed 5 new code smells, e.g., conditional complexity [6]. Refactoring can help improve software design, software understandability, and development efficiency [2]. An approach based on the quantitative analysis of the dependencies between code smells was proposed by Hamza [7], showing that some code smells require larger effort to remedy and should be concerned by developers in refactorings. Besides, the paper also presents the difference between the code smells proposed by Kerievsky and Fowler. However, in the commit records of a project, there are some special refactorings that developers explicitly admit, and such refactorings have seldom been investigated.

B. Code Smell Detection Tools

It is a complex and tedious work to detect code smells, and tools for automatic detection of code smells are beneficial to developers. Various methods and tools for automatic detection of code smells were proposed [8]. DÉCOR, a method proposed by Moha *et al*., has a good performance on specification and detection of code and design smells [8]. Tools like Klockwork, PMD, and FindBugs are the typical instances for detecting potential code errors (e.g., naming flaw) and code smells (e.g., large classes). Some tools (e.g., PMD) have been applied into software development practice.

III. STUDY DESIGN

We conducted a case study on two non-trivial OSS projects written in Java and hosted on GitHub. We follow the guidelines by Runeson and Höst [9] to describe the case study.

A. Objective and Research Questions

The goal of this study, described using the Goal-Question-Metric (GQM) approach [10], is to analyze the impact of SARs on source code for the purpose of validating its effectiveness in improving maintainability of the code, from the point of view of software developers in the context of OSS projects. We formulated two main research questions (RQs) as follows**:**

RQ1: Do SARs improve the structure quality of source code?
Rationale: The aim of refactoring is to improve the internal quality of software structure [2], thus, we want to explore

whether SARs do improve the structural quality of source code. This RQ can be further divided into two sub-RQs.

RQ1.1: Do SARs affect (introduce or reduce) code smells? If yes, which code smells are affected most frequently?

RQ1.2: Do SARs tend to introduce less code smells than non-SARs?

RQ2: What is the distribution of the severity levels of code smells affected in SARs?

Rationale: Code smell detection tools (e.g., PMD) provide code smells' severity information, which suggests the priorities of code smells for developers to fix. In SARs, the distribution of code smell severity levels can be used to assess the status of code quality.

B. Case and Unit of Analysis

According to Runeson and Höst [9], case studies can be characterized based on the way they define their cases and units of analysis. This study investigates the impact of SARs, thus we use an SAR as the unit of analysis.

For case selection, we applied the following criteria:

- The project is with a history of more than 2 years.
- The project has at least 90% of its code written in Java, since PMD is used to detect code smells and it is dedicated to identifying code smells for Java source code.
- The project has more than 20 SARs.
- The project has more than 10 committers.
- The source code of the project should be well commented (high readability and analyzability) to facilitate data analysis.

C. Data Collection

1) Data to be collected

To answer the RQs formulated in Section III.A, we collected the data items listed in TABLE I, which also lists the target RQ(s) of each data item.

TABLE I. DATA ITEMS TO BE COLLECTED

#	Data item	Description	Target RQs
D1	**NCS** – Number of Code Smells	The number of code smells of a software system of a revision (commit)	RQ1.1, RQ2
D2	**DNCS** – Delta of the Number of Code Smells	The change of the number of code smells in a commit comparing with its immediately previous commit; DNCS>0 if NCS increases, DNCS=0 if NCS does not change, and DNCS<0 if NCS decreases	RQ1.1, RQ1.2, RQ2
D3	**SLCS** – Severity Level of each newly-introduced code smell	The severity level of each newly-introduced code smell in a commit	RQ2

2) SARs Collection

For each project, we performed the following steps:

(1) Download code repository. Download the code repository of the project from GitHub.

(2) Export commit records. Export the commit records of the project using the TortoiseGit client.

(3) Identify candidate SARs. According to the definition of SAR, one way to identify SARs is to search the keywords in the commit messages of the selected projects. Fowler *et al*. defined 22 types of refactorings [2] , which can be used as basis for SAR identification. We extracted key refactoring verbs as detection roots from the refactorings, and the 22 roots of key words are shown in TABLE II. The output of this step is a set of candidate SARs.

(4) Check candidate SARs manually. Check each candidate SAR manually to exclude unexpected cases, e.g., the developer might write 'not to refactor', but no refactorings actually happened. Besides, we used a refactoring detection tool called Ref-Finder [11], to check whether refactorings had actually happened.

(5) Record commits of SARs. Record the commits corresponding to the identified SARs in a spreadsheet. Each commit contains revision number, committer, etc.

TABLE II. DETECTED KEY WORDS ROOTS

Key Words Roots
Refactor/Extract/Inline/Replace/Introduce/Rename/Move/Hide/ Encapsulate/Change/Convert/Separate/Decompose/Consolidate/ Add/Parameterize/Preserve/Pull up/Pull down/Collapse/Spilt/Substitute

3) Non-SARs Collection

To answer RQ1.2, for each selected project, it requires to collect a set of normal commits that do not contain SARs. We call such commits as non-SARs. We randomly selected a set of non-SARs, and the size of the non-SAR set equals the number of SARs in the project for eliminating the effect of quantity.

4) Code Smells Collection

We detected code smells through the PMD tool, which is a widely-used code smell detection tool adopting a static detection method. There are 33 rule sets and 237 detecting rules (e.g., Cyclomatic Complexity) in PMD. For our case study, the code repository of the selected projects were downloaded and the corresponding code snapshots to SARs of the code repository were exported. Then, we analyzed the source files of the code snapshots corresponding to each SAR and its immediately previous revision to get the differences of code smells between the two code snapshots. Fig. 2 shows the procedure of code smells collection. For each SAR or non-SAR of each selected project, we performed the following steps:

(1) Export source code. Two revisions of a project need to be exported: the revision corresponding to the SAR (or non-SAR) (V1) and its immediately previous revision (V2).

(2) Detect code smells. Use the PMD plugin for eclipse to detect code smells of revisions V1 and V2. The PMD plugin will generate reports on the detected code smells.

(3) Export code smell reports. Export the code smell reports generated in the previous step.

(4) Identify differences in code smell reports. Compare the code smell reports of V1 and V2 to identify the differences in code smells between the reports. We developed a dedicated tool to accomplish this task.

5) Data Analysis

To answer the RQs formulated in Section III.A, we need to analyze the collected data on SARs and code smells. For RQ1.1, RQ1.2, and RQ2, only descriptive statistics were used.

IV. STUDY RESULTS

A. Selected Cases

We selected two OSS projects, i.e., Fastjson [1] and Junit4 [2], as the cases. The two OSS projects are widely used in many software systems. TABLE III shows the demographic information of the three selected projects. Fastjson has 111,247 lines of code, 121 SARs, and 1,662 commits; while Junit4 has 26,579 lines of code, 28 SARs, and 2,090 commits .

TABLE III. DEMOGRAPHIC INFORMATION OF THE SELECTED PROJECTS

Project	Fastjson	Junit4
Sponsor Company	Alibaba	Apache
Access Date	5/21/2016	5/19/2016
Contributors	116	120
Line of Code	111,247	26,579
Percentage of code written in Java	99.90%	99.00%
Number of commits containing SARs	121	28
Number of commits	1,662	2,090
Percentage of SARs against total commits	7.28%	1.34%

B. Results

1) Impact on code quality (*RQ1*)

RQ1.1: TABLE IV shows the number of SARs in the two cases regarding the changed number of code smells. 70.25% (85/121) and 67.86% (19/28) of SARs did not increase code smells in Fastjson and Junit4, respectively. This suggests that more than half of the SAR revisions of the two projects were intended to decrease code smells compared with their previous versions. That is a positive signal of improving code quality.

TABLE IV. DNCS IN SELF-ADMITTED REFACTORINGS

Project	Fastjson		Junit4	
DNCS	**#(SAR)**	**%**	**#(SAR)**	**%**
DNCS > 0	36	29.75	9	32.14
DNCS = 0	39	32.23	9	32.14
DNCS < 0	46	38.02	10	35.72
DNCS ≤ 0	85	70.25	19	67.86
Total	121	100.00	28	100.00

We listed in TABLE V the top 5 code smells that are affected (introduced or decreased) most. Among all types of code smells, *DataflowAnomalyAnalysis* is the one that is affected most frequently in the two cases. Specifically, *DataflowAnomalyAnalysis* decreased in Fastjson and increased in Junit4 most frequently. This type of code smells happened most frequently as well.

TABLE V. TOP 5 MOST AFFECTED CODE SMELLS

Fastjson			Junit4		
Code smell	**DNCS**	**NCS**	**Code smell**	**DNCS**	**NCS**
DataflowAnomaly Analysis	-131	1153	DataflowAnomalyAnalysis	18	34
LooseCoupling	-24	102	SignatureDeclareThrow Exception	8	28
UnusedImports	-20	84	TooManyMethods	6	8
CyclomaticComplexity	-19	127	ConfusingTernary	3	5
SignatureDeclareThro wException	-10	702	PreserveStackTrace	1	11

RQ1.2: As shown in TABLE VI, for Fastjson, there were 121 SARs, and thus 121 non-SARs were randomly selected for comparison. The number of code smells increased (DNCS>0), kept unchanged (DNCS=0), and decreased (DNCS<0) in 76, 29, and 16 non-SARs, respectively. The number of code smells increased, kept unchanged, and decreased, in 36, 39, and 46 SARs, respectively. 37.19% (45/121) of non-SARs did not increase code smells while 70.25% (85/121) of SARs did not increase code smells in Fastjson. It means that, compared with non-SARs, SARs tend not to increase code smells in Fastjson.

However, different from Fastjson, in Junit4, more non-SARs did not increase the number of code smells than SARs. As shown in TABLE VI, 75.00% of non-SARs and 67.86% of SARs did not increase code smells in Junit4.

TABLE VI. DNCS IN SARS AND NON-SARS

Project	Fastjson		Junit4	
Name	***#(Non-SAR)***	***#(SAR)***	***#(Non-SAR)***	***#(SAR)***
DNCS > 0	76	36	7	9
DNCS = 0	29	39	11	9
DNCS < 0	16	46	10	10
DNCS ≤ 0	45	85	21	19
Total	121	121	28	28
Proportion (DNCS≤0)	37.19%	70.25%	75.00%	67.86%

2) Severity level distribution of affected code smells (*RQ2*)

PMD can detect 237 types of code smells. The priority of each code smell is defined in PMD, and it uses numbers 1 – 5 to denote the priority levels: Error High, Error, Warning High, Warning, and Information. A smaller priority number of a code smell means it is more important and more urgent to be fixed.

TABLE VII. PRIORITY DISTRIBUTION OF CODE SMELLS

PMD info		Fastjson				Junit4			
Priority	**#(PMD code smell type)**	**#(Detected code smell type)**	**Type Percentage**	**#(Detected code smell)**	**DNCS**	**#(Detected code smell type)**	**Type Percentage**	**#(Detected code smell)**	**DNCS**
Information (5)	1	1	100.00	1153	131	1	100.00	34	18
Warning (4)	14	1	7.14	84	-20	1	7.14	4	0
Warning High (3)	188	77	40.56	6169	-327	55	29.26	559	137
Error (2)	19	2	10.53	96	8	1	5.26	4	-4
Error High (1)	14	6	42.86	813	-17	3	21.43	39	10

TABLE VII shows the distribution of default severity levels of code smells identified by PMD. #(PMD code smell type) represents the number of priorities of code smells that are defined in PMD. DNCS is defined as the delta of the number of code smells in SARs. #(Detected code smell type) is the number of code smell types that were actually detected in a project. Type Percentage denotes the proportion of detected code smell types against all code smell types that can be detected by PMD. #(Detected code smell) represents the number of priorities of code smells that were actually detected by PMD in the SARs. As shown in TABLE VII, in all the SARs of Fastjson, 131 code smells with priority "Information" and 8 with priority "Error" were introduced; 20 with priority "Warning", 327 with priority "Warning High", and 17 with priority "Error High" were removed. As for Junit4, the results are similar in the distribution of code smell severity.

[1] https://github.com/alibaba/fastjson

[2] https://github.com/junit-team/junit4

V. DISCUSSION

In this section, we discuss the study results and their implications as well as threats to validity of the results.

A. Understanding on Study Results

RQ1.1: As shown in TABLE IV, more than one half of SARs tend not to increase code smells in the two projects, which indicates that the code quality is likely to be improved through SARs. It is not surprising since developers take the initiative to improve the maintainability of code in SARs. However, the results imply that not all SARs can lead to quality improvement of the source code. Only a small percentage of SARs introduced new code smells. *DataflowAnomalyAnalysis* is the code smell type that happened and affected most frequently. The top 2 largest numbers of code smells may be caused by the small granularity of code smell detection rules in PMD.

RQ1.2: From the comparison between the two datasets of SARs and non-SARs, 37.19% of non-SARs and 70.25% of SARs did not increase code smells of project Fastjson, which means less code smells introduced in SARs than in non-SARs. Nevertheless, more non-SARs than SARs did not increase code smells in Junit4, which indicates better performance of non-SARs than SARs of Junit4 in code smells reduction. We reviewed and used Ref-Finder to check the selected non-SARs, and found that refactorings had happened in some non-SARs. Additionally, the scale of Junit4 is relatively small, thus less SAR information of Junit4 than Fastjson may lead to the fluctuation of the results. Not all non-SARs do not contain refactorings and not all SARs contain refactorings. SAR is a signal to find refactorings happened, but it does not mean that refactorings definitely happened in all SARs. This indicates that some developers may not strictly distinguish refactorings from normal code changes, which may result in misunderstandings on developers' modifications on code.

RQ2: As shown in TABLE VII, the priority types of actually detected code smells cover all priority types defined in PMD. Take Fastjson for example, 42.86% of the code smells are with priority "Error High". Code smells with priority "Warning High" take the most proportion against all detected code smells, which implies that the code quality of the studied project may be moderate (similar performance in Junit4). This may partially result from the fact that the numbers of predefined code smell types are not balanced and most of code smell types of PMD are with priority "Warning High" (see TABLE VII). The number of code smells with priority "Error" increased, which is a signal of code quality sliding and should be paid special attention to.

B. Implications

More code smells means quality decline of a project, and in our case study, SARs are generally a positive sign of code quality improvement. However, some SARs may hurt the quality of source code, i.e., a SAR may not be a real refactoring. SARs indicate developers' awareness of code quality improvement, since the fact that developers claim refactorings explicitly, to some extent, represents their willingness to improve the structure quality of the code.

The distribution of severity levels of affected code smells in SARs reflects the maintainability status of a project to certain extent. When more code smells of high severity levels were removed in SARs, the project's maintainability would be more improved.

VI. CONCLUSIONS

Self-admitted refactorings (SARs) were seldom studied in previous research. In this work, we explored the SAR phenomenon from multiple perspectives. Based on the results on three studied OSS projects, we draw the following conclusions: (1) SARs tended to improve code quality, though a small proportion of SARs introduced new code smells. (2) Different projects do not have the same results on frequent-affected code smells. (3) More than half of SARs did not introduce code smells; however, non-SARs did not suggest less decrease of code smells than SARs. (4) In SARs, most code smells are with a moderate priority of "Warning High" to fix.

ACKNOWLEDGMENT

This work is supported by the National Key Research and Development Program of China (Nos. 2017YFB1400602 and 2016YFB0800401), the National Natural Science Foundation of China (Nos. 61572371, 61702377, 61472286, and 61773175), the Wuhan Yellow Crane Special Talents Program, the CPSF (No. 2015M582272), the NSF of Hubei Province (No. 2016CFB158), and the Fundamental Research Funds for the Central Universities (No. 2042016kf0033).

REFERENCES

[1] M. Abebe and C.-J. Yoo, "Trends, opportunities and challenges of software refactoring: A systematic literature review," *International Journal of Software Engineering and Its Applications,* vol. 8, no. 6, pp. 299-318, 2014.

[2] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., 1999, pp. 256-256.

[3] A. Yamashita and S. Counsell, "Code smells as system-level indicators of maintainability: An empirical study," *Journal of Systems and Software,* vol. 86, no. 10, pp. 2639-2653, 2013.

[4] A. Potdar and E. Shihab, "An Exploratory Study on Self-Admitted Technical Debt," presented at the ICSME '14 Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution, Washington, DC, USA, 2014.

[5] E. Maldonado, E. Shihab, and N. Tsantalis, "Using Natural Language Processing to Automatically Detect Self-Admitted Technical Debt," *IEEE Transactions on Software Engineering,* 2017.

[6] J. Kerievsky, *Refactoring to Patterns*. Addison-Wesley Professional, 2004, p. 400.

[7] H. Hamza, S. Counsell, T. Hall, and G. Loizou, "Code smell eradication and associated refactoring," in *European Computing Conference (ECC'08)*, Malta, 2008, pp. 102-107: Springer.

[8] N. Moha, Y. G. Gueheneuc, L. Duchien, and A. F. L. Meur, "DECOR: A Method for the Specification and Detection of Code and Design Smells," *IEEE Transactions on Software Engineering,* vol. 36, no. 1, pp. 20-36, 2010.

[9] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering,* vol. 14, no. 2, pp. 292-295, 2008.

[10] V. R. Basili, *Software modeling and measurement: the goal/question/metric paradigm*. University of Maryland at College Park, 1992.

[11] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim, "Template-based reconstruction of complex refactorings," in *2010 IEEE International Conference on Software Maintenance*, 2010, pp. 1-10.