社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

An Approach to Investigating How a Lack of Software Refactoring Effects Defect Density

Kenji FUJIWARA†, Kyohei FUSHIDA†, Norihiro YOSHIDA†, and Hajimu IIDA†

† Graduate School of Information Science, Nara Institute of Science and Technology
8916-5 Takayama, Ikoma, Nara 630-0192 Japan
E-mail: †{kenji-f,kyohei-f,yoshida}@is.naist.jp, †iida@itc.naist.jp

Abstract Refactoring is a technique for improving software design. We propose an approach to investigating how a lack of software refactoring effects defect density. In order to measure a lack of refactoring, we compute refactoring frequency by mining refactoring history from source code changes, and identify existence durations of code fragments that should be performed refactoring.

Key words refactoring, bad smell, mining software repositories

1. Introduction

Refactoring is a technique for improving software design. It is the process of changing the structure of program without changing its behavior [1]. Typical refactoring patterns are cataloged by Fowler [2]. Fowler describes that one of refactoring effects is decrease of software defects. When refactorings suppress defect introductions, performing refactorings frequently improves the maintainability of source code. In contrast, scarcely performing refactoring leaves bad codes.

In this paper, we propose a method to evaluate the necessity of frequent refactorings by investigating the effect of refactorings and the effect of lack of refactorings. In order to investigate the effect of refactoring on defect introductions, we consider two cases.

First, we consider the effect of refactorings. Fowler states developers can improve source code readability and design adaptability by performing refactoring. On the other hand, human error is a factor of defect introduction. It comes from both lack of software comprehension of developers and troublesome changes. Thus, frequent refactoring during software development seems to suppress defect introductions because it keeps the source code readable and adaptive. Hence, we investigate the following research hypothesis.

> H_1 If refactorings are performed frequently in software development, the probability of defect introductions is low.

By investigating hypothesis H_1, we confirm whether or not refactorings improve software quality from the viewpoint of defect introduction.

Second, we consider effect of lack of refactorings. When programmers scarcely perform refactoring, source code is not improved such as above. As a result, source code will become unreadable and non-adaptive in progression. Fowler defined the condition of code as "bad smell". For example, developers can use "bad smell" to judge the need of refactoring. In particular, "Long Method" and "Duplicated Code" are kinds of "bad smell". We consider that bad smells exist for the long period in source code that needs refactoring. Hence, we investigate the following research hypothesis.

> H_2 If duration of bad smell existence is long in software project, the probability of defects introduction is high.

By investigating hypothesis H_2, we confirm the effect of lack of refactoring in software development. If that hypothesis is accepted, refactoring is needed in software development(See Figure 1).

In this paper, we extract the refactoring history and the changes of code fragments detected as bad smell from source code changes recorded in Version Control System[1] (VCS)

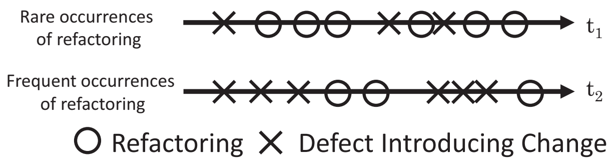
Rare occurrences of refactoring
t_1
Frequent occurrences of refactoring
t_2
○ Refactoring ✕ Defect Introducing Change

Figure 1 H_1: If refactorings are performed frequently in software development, the rate of defects inducing is low.

(1) : Version control system is the system which manages the software configuration.

such as CVS [3] and Subversion [4]. Furthermore, we identify when defects are introduced by using the information recorded in version control systems and issue tracking systems(2) (BTS) such as Bugzilla [5] and Trac [6], and then we verify H_1 and H_2.

The rest of the paper is organized as follows. Section 2 describes about bad smell and techniques of refactoring detection. Section 3 describes our evaluation method to verify H_1 and H_2. Section 4 discusses related work and Section 5 presents future works and summarizes.

2. Background

2.1 Bad smell

The criterion to judge whether performing refactoring is needed is not strictly defined. However, Fowler defined symptoms that possibly indicate a need of refactoring as bad smells. Bad smells are certain structures in the code that suggest the possibility of refactorings. For example, "Long Method" is a bad smell which suggests the "Extract Method" refactoring. In software development, developers take a lot of time to detect bad smells. In addition, it is difficult for most developers to detect bad smells because it requires skills. Therefore, a lot of methods has been developed on bad smell detection. Several of those methods are based on code clone detection techniques and source code metrics [7], [8].

2.2 Refactoring Detection

In order to analyze effects and performing of refactoring, we have to investigate when and how refactoring was performed in software development. Approaches for that are categorized into four types [9]:

Type (a) Using log messages recorded in version control systems.
Type (b) Analyzing source code changes.
Type (c) Observing developers' activity.
Type (d) Tracking usage of refactoring tools.

A type (a) approach detects evidences of refactorings by searching for the word "refactor" and possibly for related words such as "rename" or "extract" from log messages recorded in version control systems. The method assumes developers write performed refactorings into log messages in the version control system. Type (b) analyzes difference of source code between two versions. As a result, evidences of refactorings such as "Rename Field" or "Extract Method" are detected. The method allows us to detect more refactorings than type (a) because it can recover undocumented refactorings. On type (c), researchers observe how developers perform refactoring. Its range of application is narrow because it needs human resources or observation tools. In contrast, it allow us to collect detailed information of refactorings. A type (d) tracks when developers use a refactoring support feature and what part of source code is changed.

On types (a) and (b), it is possible to apply to existing software projects adopted a version control system. On the other hand, their results are not complete because they are estimation from development histories. However, type (c) and (d) cannot apply to existing projects because they require preparation.

(2): Issue tracking system is the system which centralizes defects and allows developers to track defects.

3. Investigation Method

At first, we investigate whether or not refactorings are performed frequently in target projects to verify H_1. Then, in order to verify H_2, we investigate duration of bad smell existences. Finally, we investigate the relationship between defects and these factors. Figure 2 shows an overview of our evaluation.

3.1 Refactoring Frequency

In order to verify H_1, we quantitatively evaluate whether refactorings are performed frequently. We measure *refactoring frequency* which means how many times refactoring is performed between a period of project.

We denote the set of revisions recorded in the version control system by V and each revision by v_i. Therefore, V is denoted by $V = [v_1, v_2, \cdots, v_n]$. Then, we let op_i be a source code change from revision v_i to v_{i+1}. In addition, we define the function $r(op_i)$ which returns whether op_i contains a refactoring. $r(op_i)$ returns the value one if op_i contains a refactoring, zero otherwise. Furthermore, *refactoring frequency* is defined to be $f_r(j,k)$ as follow:

$$r(op_i) = \begin{cases} 1 & (\text{ if } op_i \text{ contains a refactoring }) \\ 0 & (\text{ otherwise }) \end{cases}$$

$$f_r(j,k) = \frac{\sum_{i=j}^{k} r(op_i)}{v_k - v_j} \qquad (j < k, \quad v_j, v_k \in V)$$

In our study, in order to measure $r(op_i)$, we use the refactoring detection method based on the UMLDiff algorithm [10], [11]. It recovers design information of software, and then detects evidences of refactorings by investigating difference of the information.

3.2 Duration of Bad Smell Existence

In order to measure *duration of bad smell existence*, we first need to define bad smell in our study. Yoshida et al. proposed a method to extract "Duplicated Code" bad smell by using the code clone detection technique [8]. Miyake et al. defined a metric to evaluate the need of extract method refactoring [7]. We define bad smell as code fragment given from these two methods. Then, we measure *duration of bad*

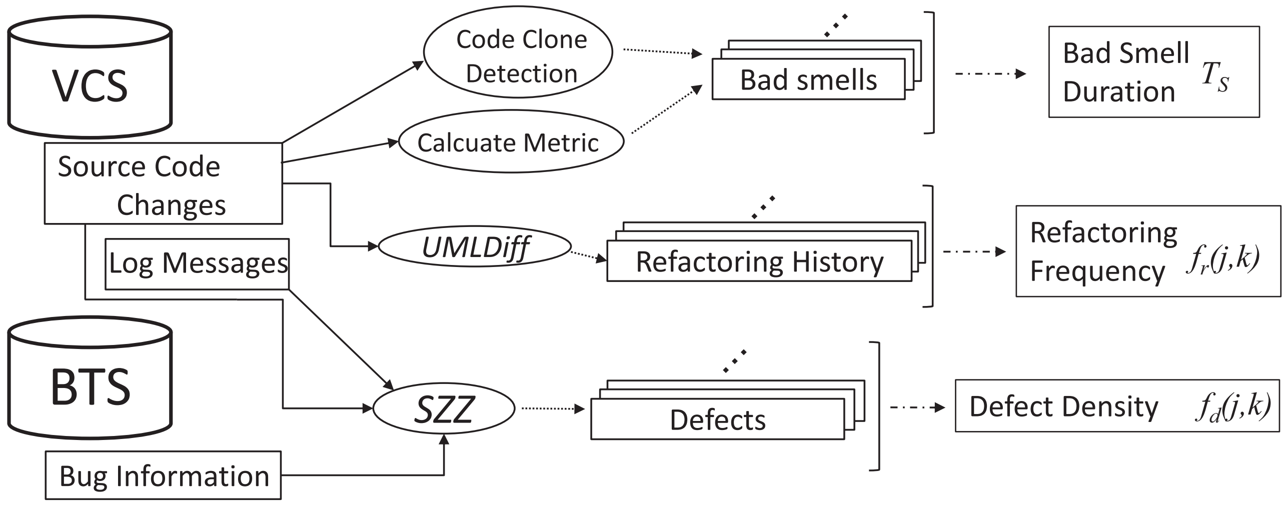

Figure 2 An overview of our research

smell existence. We denote a bad smell (*smell*) by a tuple of v_i, *classID* (a value to identify the class contains the bad smell), *beginLine*, and *endLine* (that are line number of begin or end of bad smell). The relation between revision v_i and the set of bad smells is defined as follow:

$$s(v_i) = \{smell | smell = (v_i, classID, beginLine, endLine)\}$$

Furthermore, we define historical relation between bad smells. When a bad smell $smell_a$ is contained in the revision r_i and the same code fragment $smell_b$ is contained in revision r_{i+1}, those bad smells have a historical relation. In addition, we denote *duration of bad smell existence* of set S (set of historical related bad smells).

$$\begin{aligned} T_s &= v_{sd} - v_{so}, \\ v_{so} &= \min(\{v \in V | s(v) \cap S \neq \phi\}), \\ v_{sd} &= \max(\{v \in V | s(v) \cap S \neq \phi\}) \end{aligned}$$

v_{so} is the revision which adds the bad smell, and v_{sd} is the revision of deleting the bad smell. In order to investigate historical relation between bad smells, we apply the method suggested by Kawaguchi et al [12]. Their method extract historical relation among code clones.

3.3 Defect Density

We measure *defect density* (which is the number of introductions of defect within a certain period of time). In order to measure *defect density*, we need to know when defects were introduced to the software. Śliwerski et al. designed the SZZ algorithm which is an algorithm to extract fix-inducing changes from version control systems [13]. Their algorithm links information recorded in the issue tracking system and the version control system. We consider fix-inducing changes as introductions of defect. As well as *refactoring frequency*, we define *defect density* $f_d(j,k)$ from revision v_j to v_k as follow:

$$d(op_i) = \begin{cases} 1 & \text{(if defects are introduced at } v_{i+1}) \\ 0 & \text{(otherwise)} \end{cases}$$

$$f_d(j,k) = \frac{\sum_{i=j}^{k} d(op_i)}{v_k - v_j} \qquad (j < k, \quad v_j, v_k \in V)$$

Function $d(op_i)$ returns the value one if op_i introduces defects, zero otherwise.

3.4 Requirement of Target Projects

The target project of evaluation requires to be adopted the version control system and the issue tracking system. In addition, to apply the SZZ algorithm, the project needs to collaborate with the version control system and the issue tracking system.

4. Related Work

Ratzinger et al. analyzed the relationship between refactorings and defects [14]. They used log messages in a version control system to detect refactoring. They reported when the number of refactorings is high over a certain period of the software project, defect introducing changes will decrease over successive period. Therefore, they concluded refactoring reduces defect introduction. However, they did not refer to the effect of development that continues to use low-quality source codes which requires refactoring. We evaluate the effect of lack of refactorings by using bad smells. In addition, we investigate details of time relation between refactorings and defects.

Kim et al. investigated the role of refactorings among three open source software [15]. They extracted refactoring histories from the software development history, and then evaluated the role from the following viewpoints:

- Are there more bug fixes after refactorings?
- Do refactorings improve developer productivity?
- Do refactorings facilitate bug fixes?

- How many refactorings are performed before major releases?

As a result, they reported the bug fix rate of 5 revisions before refactoring is around 26.1% for Eclipse JDT, and refactorings decrease time of bug fixes. In addition, they reported refactorings tend to be performed together with bug fixes and there are many refactoring before major releases. Their research focused on the effect of the refactorings on the bug fixes. On the other hand, our approach focused on the effect on the defect introducing changes.

In order to detect refactorings, Weißgerber and Diehl presented a method by using syntactical and signature information of the source code [16]. Their method extracts refactoring candidates by that information, and then ranks them based on code clone detection technique. Compared with the method based on UMLDiff, their method can detect less kinds of refactoring(10 kinds). In our study, we use the method based on UMLDiff because the method can detect 33 kinds of refactoring.

5. Conclusion and Future Work

In this paper, we suggest an analysis method to investigate relationship between *defect density* and *refactoring frequency* or *duration of bad smell existence*. The method evaluates the effect of performing refactorings frequently and a lack of refactorings on software development.

Currently, we are evaluating the Eclipse project(3) by our method. In addition, we are planning to evaluate other open source projects.

ACKNOWLEDGMENTS

This work is being conducted as a part of the StagE project, The Development of Next-Generation IT Infrastructure, supported by the Ministry of Education, Culture, Sports, Science and Technology. This research was also supported by Japan Society for the Promotion of Science, Grant-in-Aid for Scientific Research(No.22500027), and Grant-in-Aid for Research Activity start-up(No.22800040 and 22800043).

References

[1] M. Fowler, Refactoring: improving the design of exsiting code., Addison Wesley, 1999.
[2] M. Fowler, "Refactoring home page," http://refactoring.com/.
[3] "CVS," http://www.cvshome.org/.
[4] "Subversion," http://subversion.tigris.org/.
[5] "Bugzilla," http://www.bugzilla.org/.
[6] "Trac," http://trac.edgewall.org/.
[7] T. Miyake, Y. Higo, and K. Inoue, "A software metric for identifying extract method candidates," The Journal of the Institute of Electronics, Information and Communication Engineers, vol.J92-D, no.7, pp.1071–1073, 2009. (in Japanese).
[8] N. Yoshida, Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, "On refactoring support based on code clone dependency relation," In Proc. the 11th IEEE International Software Metrics Symposium(METRICS 2005), pp.16:1–16:10, 2005.
[9] E. Murphy-Hill, A.P. Black, D. Dig, and C. Parnin, "Gathering refactoring data: a comparison of four methods," In Proc. the 2nd ACM Workshop on Refactoring Tools(WRT 2008), pp.1–5, 2008.
[10] Z. Xing and E. Stroulia, "UMLDiff: an algorithm for object-oriented design differencing," In Proc. the 20th IEEE/ACM International Conference on Automated Software Engineering(ASE 2005), pp.54–65, 2005.
[11] Z. Xing and E. Stroulia, "Refactoring Detection based on UMLDiff Change-Facts Queries," In Proc. the 13th Working Conference on Reverse Engineering(WCRE 2006), pp.263–274, 2006.
[12] S. Kawaguchi, M. Matsushita, and K. Inoue, "Clone history analysis using configuration management system," The Journal of the Institute of Electronics, Information and Communication Engineers, vol.J89-D, no.10, pp.2279–2287, 2006. (in Japanese).
[13] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?," In Proc. the 2nd International Workshop on Mining Software Repositories(MSR 2005), pp.1–5, 2005.
[14] J. Ratzinger, T. Sigmund, and H.C. Gall, "On the relation of refactorings and software defect prediction," In Proc. the 5th Working Conference on Mining Software Repositories(MSR 2008), pp.35–38, 2008.
[15] M. Kim, D. Cai, and S. Kim, "An empirical investigation into the role of api-level refactorings during software evolution," In Proc. the 33rd International Conference on Software Engineering(ICSE 2011), pp.151–160, 2011.
[16] P. Weißgerber and S. Diehl, "Identifying refactorings from source-code changes," In Proc. the 21st IEEE/ACM International Conference on Automated Software Engineering(ASE 2006), pp.231–240, 2006.

(3) : http://www.eclipse.org/