Identifying Accurate Refactoring Opportunities Using Metrics

Yixin Bian, Xiaohong Su and Peijun Ma

Abstract Cloned code, also known as duplicated code, is among the bad "code smells." Refactoring can be used to remove clones and makes a software system more maintainable. However, there is a problem that causes the output results of the clone code detection tool cannot be directly refactored. The problem is not all the clone groups are suitable for refactoring. To address it, we propose a metric method to identify clone groups that are suitable for refactoring. The results of several large-scale software system studies indicate that our method can significantly increase the accuracy of identifying clone groups that are suitable for refactoring. It is not only beneficial to the following study of refactoring, but also it connects the entire process from clone detection to clone refactoring.

Keywords Cloned code · Metric · Refactoring

1 Introduction

Code clones are code fragments similar to one another in syntax and semantics [1]. One location is copied and paste it to another location with or without modifications during software development. This kind of activity causes multiple copies of exact or closely similar code fragments to coexist in software systems. These code fragments are known as clones [2]. In most cases, cloned codes are harmful in software

Y. Bian (✉) · X. Su · P. Ma
Department of Computer Science, Harbin Institute of Technology, Harbin, China
e-mail: bianyu79@163.com

X. Su
e-mail: sxh@hit.edu.cn

P. Ma
e-mail: ma@hit.edu.cn

maintenance and evolution [3–8]. Although code cloning can help developers to quickly reuse existing design and implementation, it also incurs a significant increase in development and maintenance cost because programmers need to apply repetitive edits when the common logic among clones changes. Furthermore, failing to apply those changes can result in defects and field failures. On the other hand, there has been a good number of empirical evidence in favor of clones concluding that clones are not harmful [9–12].

Refactoring improves code structure without changing program behavior [13]. Fowler introduced many techniques for refactoring in his book, which is widely read by practitioners [14]. One of the most frequently performed refactoring techniques is "Extract Method," which means extracting one part of an existing method as a new method and replacing the extracted part with a procedure call [15]. This technique, a common way of reducing repetitions in writing code, is also known as "extract function" or "extract procedure." The commonly used refactoring tools on various IDEs, such as Eclipse, support procedure extraction to a certain degree in order to help programmers in dealing with this common and recurring situation.

Refactoring is widely used to delay the degradation effects of software aging and facilitate software maintenance [16]. However, there is a problem that causes the output results of the clone code detection tool which is not to be directly refactored. The problem is all code clones detected by a code clone detection tool are not appropriate for refactoring [17]. So far, no study has mentioned the method of eliminating false positives of cloned code-related bugs. The chief contribution to this paper is as follows: A metric method is developed to identify clone groups that are suitable for refactoring.

The rest of the paper is organized as follows: Sects. 2 and 3 provide the background and the clone analysis algorithm developed in our research. Section 4 outlines the directions for future work.

2 Relate Work

2.1 Cloned Code

Cloned code also known as duplicated code is similar code fragments to one another in syntax and semantic. Programmers' copy–paste-modification practice is regarded as one of the main reasons for majority of clones. There are four types of cloned codes up to now:

- Type-1 clones: Identical code fragments except for variations in white-space and comments.
- Type-2 clones: Similar code snippets, where identifiers/variables can be renamed.

- Type-3 clones: Code fragments may be one or more statements added/modified/deleted beyond the syntactic similarity.
- Type-4 clones: Code fragments that perform the same calculation with different syntax.

Previous studies reported that software systems may have 5–15 % duplicated code [18], up to 50 % [19]. Based on the level of analysis applied to the source code, the techniques can roughly be classified into four main categories: textual, lexical, syntactic, and semantic [5].

2.2 The Difficulties of Identifying Refactoring Opportunities

Code clone detection can be perceived as the identification of code fragments to be refactored [3]. However, not all clone groups are suitable for refactoring. Usually, large-scale software systems have complicated intertwining logics, which makes it difficult to identify which code clones can be merged and how best to merge them [3].

3 An Approach to Identifying Refactoring Opportunities with Metrics

Other than computing resources, refactoring via function extraction incurs some software maintenance costs by resulting in dependencies. Each dependency means a contract that needs to be maintained by the development team. On the other hand, refactoring via procedure extraction also provides a benefit by resulting in a size reduction, i.e., a smaller number of code lines to maintain for the team. In this section, we derive a method to identify clone groups which are suitable for refactoring by analyzing costs and benefits of refavoring via procedure extraction. This cost–benefit analysis method makes an assumption by assigning the same weight to a dependency and a line of code. These weights can be adjusted by software developers or managers depending on their particular context and needs.

3.1 Benefits

The benefits of Extract Method refactoring are the reduction in the length of cloned code. Herein, we assume that clone group F includes code fragments f_1, f_2, …, f_m. As a result, the benefit of extracting clone group F can be represented as

$$\text{Benefit}(F) = m \times (|cf| - 1) \tag{1}$$

where |*cf*| is the number of statements which can be extracted in each fragment of group *F*. In some cases, there are some non-cloned code which cannot be moved outside the cloned code statements for the dependencies. Therefore, the statements which can be extracted may include both cloned code and non-cloned code. However, procedure extraction produces a procedure call in the original method. Therefore, actually, the length of reduction is equal to |*cf*| − 1.

3.2 Costs

Coupling is used to indicate the cost of procedure extraction. The principle of strategy for merging code clones is migration of duplicated code to another place. To migrate implemented code, it is desirable that the code has low coupling with its surrounding code [3]. In this paper, we mainly focus on data coupling. Consequently, we calculate the coupling between the original method and the new method (result of Extract Method refactoring) by counting how many parameters are needed by the new method. The detailed formula is shown as follows:

$$\text{Coupling}(F) = \sum_{i=1}^{m} \left(|P(i)_{\text{in}}|\right) + \left(|P(i)_{\text{out}}|\right) \tag{2}$$

where $|P(i)_{\text{in}}|$ and $|P(i)_{\text{out}}|$ are the amounts of the input parameters and output parameters of the new method if clone fragments are extracted from their inclosing method.

For each fragment, we denote the externally defined variables and modified by it as V_w, and externally defined variables accessed but not modified by it as V_r. The variables that appear before the fragments are denoted as V_b, and the variables that appear after the fragments are denoted as V_a. If the fragment is extracted as a new method and called in the original place, variables which appear before the fragment and accessed by the fragment (no matter read or write) should be passed in as input parameters. Those modified by the fragment and accessed by following fragments should be returned as output parameters.

The formulas are shown as follows:

$$P(i)_{\text{in}} = V_b \cap (V_w \cup V_r) \tag{3}$$

$$P(i)_{\text{out}} = V_a \cap V_w \tag{4}$$

In this paper, the $|P(i)_{\text{out}}|$ is 1 or less for we acquire the return value of the new method is no more than 1 in C programming language. If the value is more than 1, then the fragment is not suitable for extracting.

Table 1 The results of identifying clone groups that are feasible for refactoring

Products	The total clone groups (n1)	The clone groups that are feasible for refactoring (n2)	n2/n1 (%)
Linux 2.6.6/arch	5,534	4,573	82.6
Linux 2.6.6/net	2,543	1,939	76.2
Linux 2.6.6/sound/drivers	75	61	81.3
Unix/make 3.82	68	57	83.8
http2.2.2/server	121	81	66.9

3.3 Evaluation of the Benefit and Cost

The ratio of benefit/cost can be represented as

$$R(F) = \begin{cases} \frac{\text{Benefit}(F)}{\text{Coupling}(F)} = \frac{m \times (|cf| - 1)}{\sum_{i=1}^{m} (|P(i)_{\text{in}}| + |P(i)_{\text{out}}|)}, (\text{Coupling}(F) > 0) \\ \text{Benefit}(F) = m \times (|cf| - 1) \quad (\text{Coupling}(F) = 0) \end{cases} \tag{5}$$

If $R(F) > 1$. then this clone group can be suitable for refactoring or it is not.

In addition, some cloned statements are only composed of declaration statements. These cloned codes are not feasible for refactoring because of the high coupling between the original method and the new method extracted from the original one. We have evaluated all cloned code in the selected open-source programs. The results are shown in Table 1.

4 Future Work

Our results indicate that our approach accurately identify clone groups that are feasible for refactoring. In future work, we hope our study motivates IDEs such as Eclipse CDT and Microsoft Visual Studio to provide functionality to automatically analyze cloned code. We will replicate this study using more systems. In particular, we will extend our study on cloned code analysis to prune more kinds of false positives.

Acknowledgments The research is supported by the National Natural Science Foundation of China under Grant No. 61173021.

References

1. Cai, D., Kim, M.: An empirical study of long-lived code clones. In Proceedings of the 14th International Conference on Fundamental Approaches to Software Engineering: Part of the Joint European Conferences on Theory and Practice of Software, FASE'11/ETAPS'11, pp. 432–446 (2011)
2. Rahman, M.S., Saha, R.K., Krinke, J. Schneider, K.A., Mondal, M., Roy, C.K.: Comparative stability of cloned and non-cloned code: an empirical study. SAC12 March 25–29 (2012)
3. Inoue, K., Higo, Y., Kusumoto, S.: Identifying refactoring opportunities for removing code clones with a metrics-based approach. Create Space (Nov 30, 2011)
4. Juergens, E., Deissenboeck, F., Hummel, B., Wagner, S.: Do code clones matter? In: Proceedings of the 31st International Conference on Software Engineering, ICSE'09, pp. 485–495 (2009)
5. Roy, C.K., Cordy, J.R., Koschke, R.: Comparison and evaluation of code clone detection techniques and tools: a qualitative approach. Sci. Comput. Program. **74**(7), 470–495 (2009)
6. Li, Z., Lu, S., Myagmar, S., Zhou, Y.: CP-miner: finding copy-paste and related bugs in large-scale software code. IEEE Trans. Software Eng. **32**(3), 176–192 (2006)
7. Lozano, A., Wermelinger, M.: Tracking clones' imprint. In: Proceedings of the 4th International Workshop on Software Clones, IWSC'10, pp. 65–72 (2010)
8. Lozano, A., Wermelinger, M.: Assessing the effect of clones on changeability. In: IEEE International Conference on Software Maintenance ICSM, pp. 227–236 (2008)
9. Aversano, L., Cerulo, L., Di Penta, M.: How clones are maintained: an empirical study. In: 11th European Conference on Software Maintenance and Reengineering, CSMR'07, pp. 81–90, March (2007)
10. Hotta, K., Sano, Y., Higo, Y., Kusumoto, S.: Is duplicate code more frequently modified than non-duplicate code in software evolution? An empirical study on open source software. In: Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE), IWPSE-EVOL'10, pp. 73–82 (2010)
11. Kim, M., Sazawal, V., Notkin, D., Murphy, G.: An empirical study of code clone genealogies. SIGSOFT Softw. Eng. **30**, 187–196 (2005)
12. Saha, R.K., Asaduzzaman, M., Zibran, M.F., Roy, C.K., Schneider, K.A.: Evaluating code clone genealogies at release level: an empirical study. In: 10th IEEE Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 87–96 (2010)
13. Maruyama, K., Omori, T.: A security-aware refactoring tool for Java programs. In Proceedings of the 4th Workshop on Refactoring Tools, WRT'11, pp. 22–28 (2011)
14. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison Wesley, USA (1999)
15. Murphy-Hill, E., Parnin, C., Black, A.P.: How we refactor, and how we know It. In: Proceedings of the 31st International Conference on Software Engineering, ICSE'09, pp. 287–297 (2009)
16. Liu, H., Ma, Z., Shao, W., Niu, Z.: Schedule of bad smell detection and resolution: a new way to save effort. IEEE Trans. Softw. Eng. **38**(1), 220–235 (2012)
17. Ishio, T., Inoue, K., Sano, T., Choi, E., Yoshida, N.: Finding code clones for refactoring with clone metrics: a case study of open source
18. Roy, C.K., Cordy, J.R.: A survey on software clone detection research. School of Computing TR 2007-541, Queens University, pp. 115 (2007)
19. Rieger, M., Ducasse, S., Lanza, M.: Insights into system-wide code duplication. In: Proceedings of the 11th Working Conference on Reverse Engineering, pp. 100–109 (2004)