Journal of Multimedia Information System VOL. 4, NO. 4, December 2017(pp. 255-262): ISSN 2383-7632(Online)
http://dx.doi.org/10.9717/JMIS.2017.4.4.255

Which Code Changes Should You Review First?: A Code Review Tool to Summarize and Prioritize Important Software Changes

Myoungkyu Song[1] and Young-Woo Kwon[2,*]

Abstract

In recent software development, repetitive code fragments (i.e., clones) are common due to the copy-and-paste programming practice, the framework-based development, or the reuse of same design patterns. Such similar code fragments are likely to introduce more bugs but are easily disregarded by a code reviewer or a programmer. In this paper, we present a code review tool to help code reviewers identify important code changes written by other programmers and recommend which changes need to be reviewed first. Specifically, to identify important code changes, our approach detects code clones across revisions and investigates them. Then, to help a code reviewer, our approach ranks the identified changes in accordance with several software quality metrics and statistics on those clones and changes. Furthermore, our approach allows the code reviewer to express their preferences during code review time. As a result, the code reviewer who has little knowledge of a code base can reduce his or her effort by reviewing the most significant changes that require an instant attention. To evaluate our approach, we integrated our approach with a modern IDE (e.g., Eclipse) as a plugin and then analyzed two third-party open source projects. The experimental results indicate that our approach can improve code reviewer's productivity.

Key Words: Change analysis, code clone, refactoring, tool

I. INTRODUCTION

Recent research [13] has revealed that over 30% of the total amount of code is repetitive mostly because of the copy- and-paste programming practice, the framework-based development, and the reuse of same design patterns or libraries, thus creating code change patterns (i.e., refactoring or bug fixing patterns). As code changes are repetitive, anomalous changes also could repeat either by a developer's own error or by other developers' fault unknowingly.

In daily software development, it is time consuming or tedious for code reviewers to keep track of code changes all the time. In particular, because it is commonly recommended to have small commits frequently rather than having large commits, browsing individual changes in multiple revisions makes code review difficult and inefficient.

To improve the productivity of the code review process, in this paper, we introduce an enhanced code review tool that summarizes code changes including change types (i.e., refactoring types), revisions, and changed location, as well as shows which changes require more attention from a code reviewer. Additionally, our approach allows the code reviewer to express her preferences (i.e., feedback) during code review through a modern IDE (e.g., Eclipse), so that it makes possible to customize code review strategies.

First, to summarize code changes, we use code clones and an AST-based pattern matching technique. Because repetitive code fragments are likely to have potential bugs or mistakes, our tool finds all clones using a clone detection tool [9] and then examines how those clones have been evolved across revisions using pre-defined change pattern templates. Second, our tool collects change information and assesses the quality of the corresponding code using well-known software quality measurement metrics. The collected information is used to identify important code changes that require an instant attention of the code reviewer.

To demonstrate the benefits of our approach, we evaluated our approach with two third-party projects using the developed code review tool. The experimental result shows the effectiveness of our approach as our recommendation mechanism successfully informed a code reviewer that

Manuscript received December 20, 2017; Revised December 22, 2017; Accepted December 22, 2017. (ID No. JMIS-2017-0054)
Corresponding Author (*): Young-Woo Kwon, Kyungpook National University, Daegu, South Korea, +82-53-950-7566
1 Dept. of Computer Science, University of Nebraska at Omaha, USA, myoungkyu@unomaha.edu
2 School of Computer Science and Engineering, Kyungpook National University, South Korea, ywkwon@knu.ac.kr

some changes need a programmer's attention. As a result, our approach can reduce the effort of a code reviewer who has little knowledge of a code base. Overall, this paper makes the following contributions:

- Detecting and classifying code changes: Our approach can detect code changes using a change analysis platform and classify the detected changes into well-known change patterns.
- Ranking code changes: Our approach identifies important code changes that need to be reviewed first by a code reviewer based on software quality metrics, change statistics, and user feedback.
- Empirical evaluation: We evaluate our approach by conducting assessments on two third party projects and user studies.

The rest of this paper is organized as follows. Section III presents our approach. Section IV empirically evaluates our approach. Section V compares our approach and other closely related approaches, and then we conclude this paper in Section VI.

II. MOTIVATING EXAMPLE

In this section, we describe an example that motivates this research. Suppose a software development team decided to perform refactoring on their project to improve software quality regarding readability and maintainability. In particular, Alice applied the `Extract Method` refactoring to multiple cloned regions to remove duplicated code fragments. She created a new method with the duplicated code fragments and then manually replaced all the occurrences of the duplicated code fragments with the newly created method. Moreover, Bob applied the Move Type To New-file refactoring to an inner class to make it as a general, reusable type. The other team members also applied different refactoring practices [7]. Assume Carly is a project manager and in charge of reviewing a code base once a week. To ensure that there is no mistake in these refactorings, Carly needs to investigate line level differences file by file, which is usually omission-prone because Alice made changes in multiple files across revisions. As a result, Carly needs tool support to inspect the Alice's changes separately from other irreverent issues.

In addition, Carly may want to only review the code fragments affected by the Move Type To New-file refactoring and look at similar changes together because based on her experience she knows junior programmers often make mistakes when applying this refactoring practice. In other words, she may want to prioritize code change inspection tasks based on her own criteria, so that she can efficiently review all code changes. To that end, our approach helps developers (or programmers depending your context) selectively inspect code changes based on the level of the change severity during peer code reviews.

III. APPROACH

In this section, we detail our approach, a code review tool that can identify important code changes to help code reviewers.

3.1. Approach Overview

Figure 1 shows our approach overview. Our approach can be mainly summarized in two steps—change analysis and review recommendation. Specifically, the first step is to analyze a project repository using three analysis tools including a commit analyzer, a code quality measurement tool, and a clone detection tool. The clone detection tool takes as input revision history and finds code clones across revisions. The change analysis engine analyzes how those code clones have been changed across revisions and then determine their change types using a set of pre-defined change pattern templates.

The latter part of our approach is to identify important changes that require more attention from a code reviewer. The recommendation engine takes the identified change

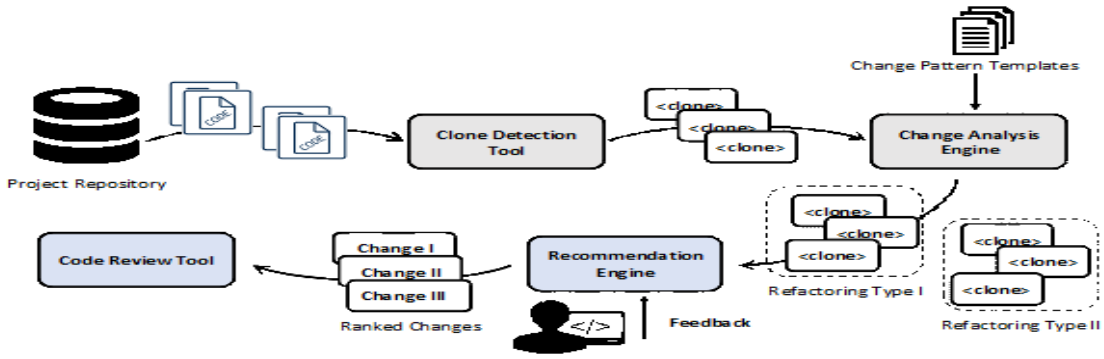

Fig 1. Approach overview.

types from the change analysis engine as well as feedback from the code reviewer. Then, those changes are rearranged in accordance with reviewer's preferences and the severity of changes. In the following sections, we describe our approach in detail.

3.2. Identifying Important Code Changes

Our approach analyzes code changes using a set of code assessment tools including commit-based analysis, clone-based analysis, and pattern-based analysis. In the following discussion, we describe each analysis tool in turn.

3.2.1 Pattern-based Change Analysis

In this section, we dis- cuss how changes are analyzed using pre-defined patterns. Our approach analyzes the subsequent revisions ($r_{i+1}, r_{i+2}, ..., r_n$) of the original clone c_i, and then identifies ASTs that are related to the clone c_i to see how a particular clone (i.e., code change) evolves across revisions. By comparing the clone c_i and the evolved clone c_j, it makes possible to infer its change pattern, which is a refactoring type. Specifically, our approach is template-based pattern matching that consists of pre- and post-edit matchers. We implemented six pattern matching templates based on well-known refactoring practices [7] as follows:

- **Extract Method**: turns code fragments into a method.
- **Pull-up Method**: moves code fragments to a super class.
- **Extract Super-class**: creates a super class and moves common methods or variables to the super class.
- **Move Type To New-file**: creates a new compilation unit for the selected member type (e.g., inner class), updating all references as needed.
- **Extract and Move Method**: turns code fragments into a method and then moves the extracted method to other class.
- **Extract and Pull-up Method**: turns code fragments into a method and then moves it to a super class.

Finally, we obtain a list of frequently occurring refactorings as follows: $r_j \in R = \{r_{i+1}, r_{i+2}, ..., r_m\}$. Changes that are not classified into the above refactoring types are tagged undefined (UD). All this information is passed to the review assistance tool to infer important refactorings that need to be reviewed first.

3.2.3 Clone-based Change Analysis

To identify code change patterns, our approach uses a clone detector and matches changes of code clones with our predefined templates. The clone detector, Deckard—a tree-based clone detection tool [9], results in clone groups (CG) which contains a set of clones. Then, we repeat the same process for the particular revision range where a code reviewer must review. The found clone groups are partitioned based on similar changes, which are considered as potential repeated code changes. Based on those code changes, we determine their change types (i.e., refactoring types) using our AST-based pattern matching tool. Because found clones are only pieces of a code base which are syntactically incomplete, it is inaccurate to find specific change patterns based on code fragments. Thus, we find ASTs that contain each code clone and the ASTs will be compared with predefined change patterns.

3.3. Recommending Code Review

To help code reviewers better understand code changes across revisions and reduce review efforts, our code review tool recommends review strategies by showing important changes that need to be reviewed first at code review time. To that end, identified changes (i.e., refactorings and unclassified code changes) are reordered based on software quality metrics, statistics on changes, and feedback provided by a code reviewer. In this section, we describe two recommendation models—statistics- and feedback-based recommendation.

3.3.1 Statistics-based Recommendation

The first recommendation model uses static information including software quality metrics and statistics on code changes. In particular, ranking scores for the recommendation can be calculated using the following metrics: (1) uncommented lines of code (LoC), (2) McCabe's cyclomatic complexity (CC), (3) the weighted number of methods in a class (WNC), (4) the occurrence of a same code change types (OC), and (5) the number of clones in a same change group (NC).

The quality of code fragments containing each change $c_j \in C$ is assessed at micro-level and macro-level. In particular, we measure the lines of code and cyclomatic complexity for each code change. The lines of code pertain to the lines of code common to the clones and the cyclomatic complexity is calculated for the specific method involved in the change. For the cyclomatic complexity, we first calculate the cyclomatic complexity for each method covering a particular code change and then choose the maximum complexity of them. These two metrics can provide better understanding about code changes made by programmers at micro level.

To evaluate code changes at macro level, our approach uses statistics on code changes. Specifically, we count the weighted number of methods in a class, which is not just a simple count of methods in a class but a metric which is the sum of the complexities of all methods of

the class pertaining to the change [12]. Furthermore, we calculate the occurrence of a same code change type, which represents how frequently a particular refactoring occurred in the given revision range.

For example, if `Extract Method` is the most frequently performed refactoring across revision, the code reviewer can review all `Extract Method` refactorings. By reviewing similar refactoring patterns at same time, the code reviewer may reduce the overall code review time. Finally, the number of clones in a same change group represents how often the same refactoring was repeated. In particular, when a programmer simply copies and pastes a particular piece of code, there are a number of clones in a same change group. Because repetitive code fragments are highly vulnerable to evolution, those changes need to be first reviewed.

The final statistics-based ranking score, s_i is calculated as follows:

$$s_i = \frac{cc_i + loc_i}{2} + \frac{wnm_i + f_i + n_i}{3}$$

where, *cci* is McCabe's cyclomatic complexity; wnmi is the weighted number of methods; loci is lines of code; fi is the occurrences of a same code change type; OC and ni are the number of clones in a same change group. All the values of CC, W NM, LoC, OC and NC are normalized ranging between 0 and 1, and then the final ranking score is between 0 and 2.

3.3.2 Feedback-based Recommendation

In addition to the aforementioned metrics and statistics, we take a code re- viewer's feedback into consideration because different code reviewers have different review strategies. For example, one reviewer may want to review the code changes made by a particular programmer. Another reviewer may be only interested in a particular refactoring type that are potentially based on his or her previous experiences. To that end, we first group similar clones using the following attributes:

- **Clone group**: similar clone changes are grouped into the same clone group.
- **Refactoring type**: clone groups are classified into well-known refactoring types described above. If there is no matched refactoring type for the clone group, they belong to a unclassified group.
- **Revision number**: if different code clones have a same revision number or close revision numbers, these changes might occur at the same time.
- **Package**: if code changes have the same package name, they are closely related.
- **Programmer information**: if programmer information is provided, clones created by the same programmer are grouped together.

Then, to include user feedback in the ranking, we allow code reviewers to express their feedback through a code review tool integrated and then recalculate their rankings as followings:

$$\forall d_j \in C, d_j = RT + RV + PK + CG$$

where, RT = 0.25 when refactoring type of the change $c_j \in C$ in the ranked list is same as that of the Up-Voted change. Similarly, values of RV, PK and CG are set to 1 when type, revision, package and clone-group respectively of the changes in the ranked list is same as that of the Up-Voted change. Otherwise the values are set to 0. The idea is to dynamically cluster changes that the reviewer intends to move them up the ranking order for further inspection.

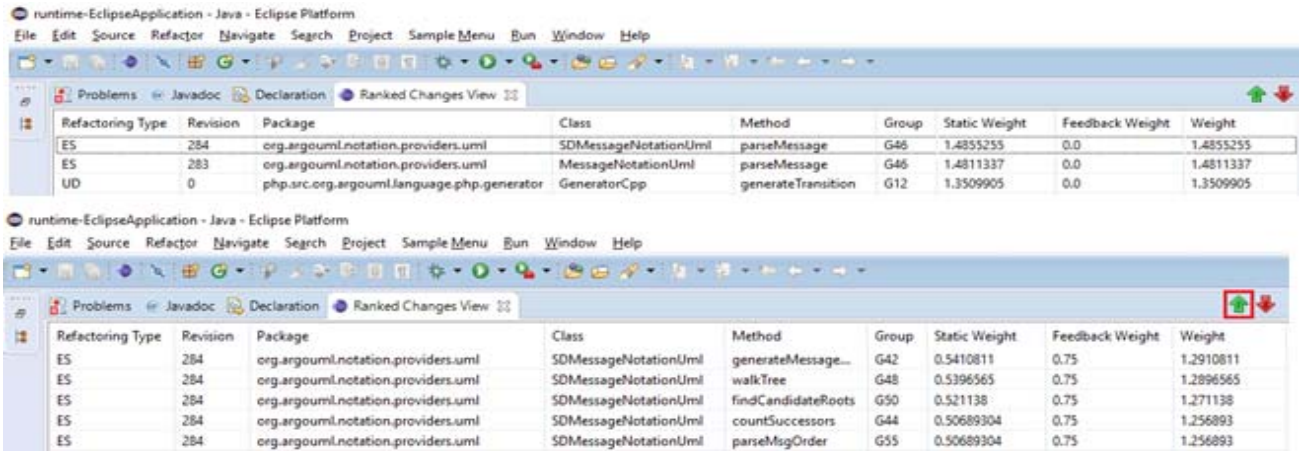

Fig. 2. Top: initial recommendations based on program quality metrics and statistics; Bottom: ranking changes according to user feedback.

IV. EVALUATION

In this section, we evaluate our approach and tool through two third-party projects—ArgoUML and Apache Tomcat 2. We show how our tool presents changes to a code reviewers in accordance with several different metrics and user feedback.

4.1. Code Review Recommendation

Next, we evaluate our recommendation engine with the ArgoUML project. We first show an initial recommendation result computed only using static data and then show how those rankings change in accordance with user feedback.

4.1.1 Statistics-based Recommendation

Figure 2 shows the screenshot of the ranked code changes in Eclipse. On the manual inspection of the top two changes, it was found that the `parseMessage` method was extracted from both SDMessageNotationUml and MessageNotationUml and placed in the super class AbstractMessageNotationUml subsequently accross revisions 283 and 284. The extracted parseMessage method seems to be highly ineffecient because of its longer LoC (702) and higher complexity than other code changes. Surprisingly, the comment made by a developer about the method— *"TODO: - This method is too complex, lets break it up"*, *"@throws ParseException when it detects an error in the attribute string. See also ParseError.getErrorOffset()"*—, and other comments indicate that this class and method indeed needs more attention from a code reviewer.

4.1.2 Feedback-based Recommendation

Finally, we evaluate our feedback-based recommendation mechanism by emulating user feedback. A code reviewer can express their preferences by simply pressing the up- or down-button at the IDE, resulting in the differently ordered code changes. Figure 2 (bottom) shows the screenshot of the changed list of code changes. In this example, we pressed the `Up-Vote` button (i.e., the green button in Figure 3) for the `Extract Method` refactoring performed in the `SDMessageNotationUml` class. Then, similar changes (e.g., same refactoring type, same package and class, same clone group) should move up in the code review tool. As illustrated in Figure 3 the tool dynamically clustered and moved up the ranked list, six changes of the `Extract Method` refactoring type with the same class and revision but different methods based on the given feedback. The code reviewer in this instance can now review six different changes that are all related to each other, at once because they were grouped together.

V. RELATED WORK

The presented approach is closely related to clone detection and change analysis techniques. To the best of our knowledge, a code review tool using code clones and their change patterns is the first of its kind. Thus, in this section, we briefly introduce representative research efforts for clone detection and change analysis techniques and compare our code review tool with state-of-the-art code review tools and refactoring tools.

5.1. Clone Detection and Change Analysis

In the software engineering community, clone detection techniques have been widely discussed for the last decade. Göde and Koshke [8] and Nguyen et al. [14] introduced an incremental clone detection algorithm analyzing the results of previous versions. Krinke [11] detected code clones in five open source systems and studied how clone groups had been consistently changed. Saha et al. [15] and Aversano et al. [4] studied how clones are evolved. Kim et al [10] also studied the evolution of clones and classified evolving code clones. We leverage those research efforts to detect code clones and understand how they have been evolved across revisions to infer their change patterns.

Xie et al. found challenges for code change comprehension and a lack of tool support for understanding composite changes [16]. Recent research focuses on identifying or rank- ing refactoring candidates [17], [6]. Our primary goals are to detect repetitive code changes and then rank them based on multiple software quality metrics, change statistics, and dynamic user feedback.

5.2. Code Review and Refactoring Tools

Existing code review tools [1], [2], [3] are usually used in practice but require exploring each line by manually browsing files. Even though cross-file changes are made with code clones, programmers must manually find all the locations that were changed using a similar refactoring practice. Unlike state- of-the-art code review tools, our tool enables a code reviewer to take a look at all similar changes at a time.

Another research efforts are to identify refactoring candidates [5], [18]. Balazinska et al. [5] classify clone groups, measuring their differences based on a clone classification scheme, and provide refactoring

opportunities. Tsantalis et al. [18] use a program slicing technique to capture code modifying an object state and design rules to identify refactoring candidates from slices. While these approaches focus on identifying refactoring opportunities, our approach focuses on identifying and ranking refactorings examples.

VI. Conclusion

In this research, we explored how code reviewers can be assisted with a tool to identify important code changes. Our code review tool can detect code clones and classify them as meaningful code changes (e.g., refactoring types). Furthermore, our tool can recommend code review strategies to a code reviewer. Through this study, we evaluated our approach through third party open source projects. The experimental results indicate that our code review tool integrated with a modern IDE can effectively identify important code changes and classify them into refactoring types, thereby improving the productivity of a code reviewer.

As a future research direction, we will create more change pattern templates to find other refactoring types as well as bug repair activities. Also, we will extend our tool to detect potential mistakes occurred during refactoring and then guide a code reviewer with a set of possible solutions with real examples to correct such mistakes.

Acknowledgement

This work was supported by the National Research Found ation of Korea(NRF) grant funded by the Korea governme nt(MSIT) (No. 2017R1C1B5075658).

REFERENCES

[1] Code collaborator: https://smartbear.com/product/collaborator/, 2017.

[2] Gerrit, http://code.google.com/p/gerrit/, 2017

[3] Phabricator, http://phabricator.org, 2017

[4] L. Aversano, L. Cerulo, and M. D. Penta, "How clones are maintained: An empirical study," In *CSMR '07: Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, 2007.

[5] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis, "Measuring clone based reengineering opportunities," In *Proceedings of Software Metrics Symposium. Sixth International*, pp. 292–303, 1999.

[6] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis, "Partial redesign of Java software systems based on clone analysis," *In 6th Working Conference on Reverse Engineering*, 1999

[7] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 2000.

[8] N. Göde and R. Koschke, "Studying clone evolution using incremental clone detection," *Journal of Software: Evolution and Process*, vol. 25, no. 2, pp. 165–192, 2013.

[9] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard, "Scalable and accurate tree-based detection of code clones," *In Proceedings of the 29th International Conference on Software Engineering*, 2007.

[10] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," *In Proceedings of the 13th International Symposium on Foundations of Software Engineering*, 2005

[11] J. Krinke, "Is cloned code more stable than non-cloned code?," *In Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, 2008.

[12] R. Moser, A. Sillitti, P. Abrahamsson, and G. Succi, "Does refactoring improve reusability?," *In International Conference on Software Reuse*, 2006

[13] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, and H. Rajan, "A study of repetitiveness of code changes in software evolution," *In Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, 2013.

[14] T. T. Nguyen, H. A. Nguyen, J. M. Al-Kofahi, N. H. Pham, and T. N. Nguyen, "Scalable and incremental clone detection for evolving software," *In IEEE International Conference on Software Maintenance*, 2009.

[15] R. K. Saha, C. K. Roy, K. A. Schneider, and D. E. Perry, "Understanding the evolution of type-3 clones: an exploratory study," *In Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, 2013.

[16] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim, "How do software engineers understand code changes?: An exploratory study in industry," *In Proc. of FSE*, 2012

Journal of Multimedia Information System VOL. 4, NO. 4, December 2017(pp. 255-262): ISSN 2383-7632(Online)
http://dx.doi.org/10.9717/JMIS.2017.4.4.255

Authors

Young-Woo Kwon is an assistant professor at the School of Computer Science and Engineering at Kyungpook National University. Prior to coming to KNU, he was an assistant professor at the Department of Computer Science at Utah State University. He received his PhD in Computer Science in 2014 from Virginia Tech. His research interests span Mobile Computing, Cloud-Based Systems, and Software Engineering, as applied to Middleware, Energy Efficiency, and Software Refactoring.

Myoungkyu Song is an assistant professor at the Computer Science Department at the University of Nebraska at Omaha since 2015. Prior to coming to UNO, he was a postdoc in the Center for Advanced Research in Software Engineering (ARiSE) at the Department of Electrical and Computer Engineering at the University of Texas at Austin. He received his Ph.D. in Computer Science in May 2013 from Virginia Tech. One of his chief research interests is programmer productivity, which spans the spectrum from software engineering to program analysis, addressing related issues to make it easier to develop, maintain, and evolve large scale software systems.