Code Clone Detection & Removal Using Class Hierarchy Based Segment Mapping With Adaptive Refactoring

Pooja kapila[1], Amitabh sharma[2], Navleen kaur[2]
[1]Chandigarh Engineering College,Landran ,India .
[2]Associate Professor, Chandigarh Engineering College,Landran ,India

Abstract- The clone detection is made between the various clones of same number of lines of code. The small or sub-clones in the longer programs or functions than the evaluated code chunk are usually neglected and returned with no matching results. If some programmer adds a new lines of code with optional parameters in the end of the function which matches with a shorter function can with stand in the code by using the existing scheme. Hence it can be called that the existing method is enough capable of detecting the clones in non-optimal and optimal paradigms, but it does not evaluate the third case where the extra-optimal case has been employed. In case a third condition is called in the code clone which is making its structured unique than the existing method declaration, can be also combined by adding the third condition in the bottom to sum up the code in single segment.
Keywords- Clone detection, Code clone, optimal clones, non-optimal clones.

I. INTRODUCTION

Code clones have received great interests in recent years from many researchers, engineers, and practitioners in the field of software engineering. A code clone is defined as a group of code fragments that are identical or similar to one another. Code clones are introduced into source code of software systems by various reasons, and the most typical one is code cloning by copy-and-paste operations for reusing existing features. Typical software systems contain a certain amount of code clones because code cloning is a common practice for software developers. The existence of code clones has been regarded as a bad smell for software evolution over a period of time because code clones require much attention to be maintained. Once code clones are introduced into source code, most of them should be consistently maintained. Unintended inconsistencies among code clones have a high risk for introducing bugs in software systems. However, it is not an easy task for developers or maintainers to be aware of all the code clones and maintain all of them consistently, specifically in the case of large software systems. This is a reason why code clones are regarded as bad factors for software evolution. Many researchers have proposed a variety of techniques to cope with code clones based on this common wisdom. However, some of recent empirical studies have been against it. That is, these studies revealed that code clones do not highly affect software evolution. The discussion for harmfulness of code clones remains inconclusive, but it is widely accepted that not all but a part of code clones have negative impacts on software evolution. For these reasons, it is not effective to prohibit software engineers from code cloning. Furthermore, prohibiting code cloning is also unrealistic because of advantages of it. Therefore, it is strongly required to manage code clones effectively. The objective of the work described in this dissertation is to promote efficient software evolution through effective managements for code clones. To achieve this objective, it is necessary to know state-of-the-art of research achievements. Therefore, a survey was conducted on research literatures on code clones. This survey categorized literatures into five categories, detection, removal, prevention, analysis, and bug detection. The survey told the current states of re-search on code clones. It also told us limitations of previous research on code clone analysis, and unclear points of characteristics of them.This study detected clone genealogies, which represent how individual code clones evolved, and analyzed them. This investigation was interested in how long code clones survived and how many times they were modified throughout their lifetimes. The experimental results reaffirmed that most of clones had short lifetimes and were modified at most once, both of which were reported in previous research. Furthermore, the results revealed some characteritics that have not been revealed in previous research. One of the findings is that approximately 3% of code clones are long-lived and modified multiple times. In other words, approximately 97% of code clones do not require high costs to be maintained. This finding empirically supports an opinion that not all but a part of clones affect software evolution. Another finding is that code clones tend to be modified more frequently in the former halves of lifetimes than in the latter ones. This finding suggests that it is necessary to start managing code clones in earlier stages of their lifetimes for an effective code clone management. In the next, we proposed a way to cope with code clones, which is a support to remove them with a particular refactoring pattern. The refactoring pattern used in this study enables to remove code clones even if they include some gaps. Moreover, the proposed technique performs a fine-

grained analysis on source code, which allows to handle instances that any of previous refactoring supports cannot handle. Furthermore, the proposed technique can handle all the code fragments included in a code clone as against to existing techniques that can only handle a pair of code fragments. A software tool was developed as an implementation of the proposed technique, and validated the usefulness of the technique through two experiments, one is on open source software systems, and the other is with subjects.

II.TYPES OF CODE CLONES

"Textual Similarity" and therefore fourth one is understood as "Functional Similarity". A code fragment that has identical or similar code fragment(s) to within the source code, in general, terms as code clone. Copying code fragments and these are reuse by pasting with or without modifications are very common activities in software development. This type of re-use approach of existing code is called the code cloning and the pasted code fragment is called a clone of the original . The process is called the software cloning. The clones are those segments which are used according to some definition of similarity. However, in software engineering field, the term code clones is still searching for a suitable definitions. There are also many other software engineering tasks such as understanding code quality, aspect mining, plagiarism, software evolution analysis etcetera. Code clones are considered as bad smells of the software system . Moreover, clones are also the result of copy paste activities. Clones are believed to possess a negative impact on evolution. More discussion on the reason for cloning can be found elsewhere. There are 4 types of the clones. First three are called the

Textual Similarity:

Type I: (Exact clone) here are the clones in which variations in white spaces and comments.

Type II: (Renamed) program fragments which are structurally similar expects for changes in identifiers, literals, types and comments.

Type III: (Near miss) these are clones that have copied with further modifications like changes in identifiers, variables etcetera.

Type IV: (Semantic clones) programs fragments are functionally similar without being textually similar.

III. CLONE DETECTION TECHNQIUES

Copy code is a computer programming term for an arrangement of source code that happens more than once, either inside a program or crosswise over various programs owned or kept by the same entity. Duplicate code is for the most part viewed as undesirable for various reasons. A minimum requirement is usually applied to the quantity of code that must show up in a sequence for it to be considered duplicate rather than coincidentally similar. Sequences of duplicate code are sometimes known as code clones or just clones; the automated procedure of discovering duplications in source code is called clone detection.

Various clone detection techniques are referred within in the literature.

Text Based Technique: There's several clone detection techniques that are based on pure text-based strategies. In this, fragments are compared with one other to search out sequence of same text. Text based technique compare the text, line by line, with or without normalization the text by renaming the identifiers, filtering out the comments and variations within the layout.

Token Based: In this approach, the source system is parsed or transformed in to the sequence of tokens. After that, the tokens are scanned for finding the duplicate code and these duplicated sequence returned as clones.

Tree Based: During this approach, the program is split into parsed tree with a parser of the language of interest. Further, the sub trees are searched with some tree matching techniques and therefore the corresponding code of the same tree are reported as clone pairs.

PDG Based: Program Dependency Graph (PDG) is one step further in accruing a source code representation of high abstraction than alternative approaches by considering the semantic information of the source. PDG contains the control flow and data flow information of the program and therefore the code corresponding to the sub graph are identified and represent as derived (copied) code (clones).

Metrics Comparison Based Technique: To detect the function clone, Metrics based technique is employed. There are numerous clone detection techniques that use totally different metrics to realize the similar code. A fingerprinting (a set of software's) functions are work out for syntactic units (a class, a function or a method) and then the value are compared to search out the copied code over original code.

Tracking Clipboard Operations: This technique is relies on the copy-paste activity of the programmers for the new creation of the clones. The fundamental result of this clipboard operation is the simply clipboard activities in the editor (inside the IDE such as Eclipse). When the code segment pasted after copy this copied segments are recorded as clone pairs.

IV. LITERATURE REVIEW

Tsangaris, Nikolaos et. Al [1]has worked on assessing the Refactorability of Software Clones. The presence of duplicated code in software systems is significant and several studies have shown that clones can be potentially harmful with respect to the maintainability and evolution of the source code. Despite the significance of the problem, there is still limited support for eliminating software clones through refactoring, because the unification and merging of duplicated code is a very challenging problem, especially when software clones have gone through several modifications after their initial introduction. In this work, an approach was proposed for automatically assessing whether a pair of clones can be safely refactored without changing the behavior of the program. In particular, our approach examines if the differences present between the clones can be safely parameterized without causing any side-effects.

Roy, Chanchal K. et. al. [2] has worked on studying the the vision of software clone management: Past, present, and future. This paper presents a comprehensive survey on the state of the art in clone management, with in-depth investigation of clone management activities (e.g., tracing, refactoring, cost benefit analysis) beyond the detection and analysis. This is the first survey on clone management, where point is to the achievements so far, and reveal avenues for further research necessary towards an integrated clone management system. The authors have believed that we have done a good job in surveying the area of clone management and that this work may serve as a roadmap for future research in the area

Tufano, Michele et. al. [3] has surveyed that when and why your code starts to smell bad. There are several factors that contribute to technical debt. One of these is represented by code bad smells, i.e., symptoms of poor design and implementation choices. While the repercussions of smells on code quality have been empirically assessed, there is still only anecdotal evidence on when and why bad smells are introduced. To fill this gap, the research have conducted a large empirical study over the change history of 200 open source projects from different software ecosystems and investigated when bad smells are introduced by developers, and the circumstances and reasons behind their introduction. Their study required the development of a strategy to identify smell introducing commits, the mining of over 0.5M commits, and the manual analysis of 9,164 of them (i.e., those identified as smell introducing).

Kim, Miryung et. al. [4] has worked on the empirical study of refactoringchallenges and benefits at microsoft. This paper presents a field study of refactoring benefits and challenges at Microsoft through three complementary study methods: a survey, semi-structured interviews with professional software engineers, and quantitative analysis of version history data. Our survey finds that the refactoring definition in practice is not confined to a rigorous definition of semantics-preserving code transformations and that developers perceive that refactoring involves substantial cost and risks. The author s have also report on interviews with a designated refactoring team that has led a multiyear, centralized effort on refactoring Windows. The quantitative analysis of Windows 7 version history finds the top 5 percent of preferentially refactored modules experience higher reduction in the number of inter-module dependencies and several complexity measures but increase size more than the bottom 95 percent.

Koschke, Rainer et. al.[5] has developed the software clone management towards industrial application. This report documents the program and the outcomes of Dagstuhl Seminar 12071 "Software Clone Management Towards Industrial Application". Software clones are identical or similar pieces of code or design. A lot of research has been devoted to software clones. Unlike previous research, this seminar put a particular emphasis on industrial application of software clone management methods and tools and aimed at gathering concrete usage scenarios of clone management in industry, which will help to identify new industrially relevant aspects in order to shape the future research.

Zibran, M., and C. Roy [6] has defined the t*he road to software clone management in the survey study*. With regards to the scheduling techniques, the evolutionary algorithms such as GA as well as the artificial intelligence (AI) techniques such as heuristic based approaches may suffer from local optima, and do not guarantee optimality.

O'Keele et al. conducted an empirical comparison of simulated annealing (SA), GA and multiple ascent hill-climbing techniques in scheduling refactoring activities in five software systems written in Java. They reported that among those AI techniques, the hill-climbing approach performed the best. CP is a relatively recent technique that combines the strengths of both AI and OR techniques and thus can be expected to perform better. Nonetheless, an empirical comparison of CP with AI and evolutionary algorithms in optimizing the scheduling of code clone refactoring can be an interesting study.

V. PROPOSED WORK OVERVIEW

The clone detection is made between the various clones of same number of lines of code. The small or sub-clones in the longer programs or functions than the evaluated code chunk are usually neglected and returned with no matching results. If some programmer adds a new lines of code with optional parameters in the end of the function which

matches with a shorter function can with stand in the code by using the existing scheme. Hence it can be called that the existing method is enough capable of detecting the clones in non-optimal and optimal paradigms, but it does not evaluate the third case where the extra-optimal case has been employed. In case a third condition is called in the code clone which is making its structured unique than the existing method declaration, can be also combined by adding the third condition in the bottom to sum up the code in single segment. The existing method is enough capable of detecting the clones in non-optimal and optimal paradigms, but it does not evaluate the third case where the extra-optimal case has been employed. In case an third condition is called in the code clone which is making its structured unique than the existing method declaration, can be also combined by adding the third condition in the bottom to sum up the code in single segment. The existing refactoring method relies upon the control flow graphs of individual code fragments. A full mapping tree can be employed over the whole code to refactor the code with more intelligence.

VI. METHODOLOGY

This research project will start with a detailed literature review on the various clone detection and management and refactoring schemes. Then, a detailed coverage of the code detection and management techniques designed to prevent the issue of non-optimal and optimal code clones in the source codes of various tools. The simulation would be implemented using Eclipse or Netbeans tools for java. The obtained results would be examined and compared with the existing code clone detection and mechanism to address the similar issues. Waterfall development method is ideal for projects with clear task formalization and fixed scope of work like this research work, i.e. for small and medium-size projects.

VII. CONCLUSION

The existing model is created to detect the code clone fragments with body of the same method or other methods and duplicate method declarations somewhere in the code files. The existing method recognizes the variability in member variables but not in the procedure body variables. The existing model relies upon the exact clone detection when it comes to the procedure clone detection and does not evaluate the changed variable or other entity declarations. Two methods, which are having the exactly same flow but declared with different variable entities, are also considered clones. The certain improvement must be made in order to evaluate such clones also. The existing method is enough capable of detecting the clones in non-optimal and optimal paradigms, but it does not evaluate the third case where the extra-optimal case has been employed. In case an third condition is called in the code clone which is making its structured unique than the existing method declaration, can be also combined by adding the third condition in the bottom to sum up the code in single segment. The existing refactoring method relies upon the control flow graphs of individual code fragments. A full mapping tree can be employed over the whole code to refactor the code with more intelligence.

VIII. REFERENCES

[1] N. Tsantalis, D. Mazinanian, and G. P. Krishnan, "Assessing the Refactorability of Software Clones," *IEEE Trans. Softw. Eng.*, vol. 41, no. 11, pp. 1055–1090, 2015.

[2] C. K. Roy and R. Koschke, "The Vision of Software Clone Management : Past , Present , and Future," 2007.

[3] M. Tufano *et al.*, "When and why your code starts to smell bad," *Proc. - Int. Conf. Softw. Eng.*, vol. 1, pp. 403–414, 2015.

[4] M. Kim, T. Zimmermann, and N. Nagappan, "An empirical study of refactoring challenges and benefits at Microsoft," *IEEE Trans. Softw. Eng.*, vol. 40, no. 7, pp. 633–649, 2014.

[5] R. Koschke, I. Baxter, M. Conradt, and J. Cordy, "Software Clone Management Towards Industrial Application (Dagstuhl Seminar 12071)," *Dagstuhl Reports*, vol. 2, no. 2, pp. 21–57, 2012.

[6] "The University of Saskatchewan Department of Computer Science Technical Report # 2012-03 The Road to Software Clone Management :," 2012.