MULTI-STEP AUTOMATED REFACTORING FOR CODE SMELL

M.Lakshmanan[1], S.Manikandan[2]

[1]Assistant professor, [2]PG Scholar, Department of CSE, Gnanamani College of Engineering, Tamilnadu, India

Abstract

In computer programming, code smell may origin of latent problems in source code. Detecting and resolving bad smells remain time intense for software engineers despite proposals on bad smell detecting and refactoring tools. Numerous code smells have been recognized yet the sequence in which the detection and resolution of different kinds of code smells are performed because software engineers do not know how to optimize sequence. In this paper, the novel refactoring approach is proposed to improve the performance of programs. In this recommended approach the code smells are automatically detected and refactored. The simulation results propose the reduction of time over the semi-automated refactoring are achieved when code smells are refactored by using multi-step automated refactoring.

Keywords: *Code smell, multi step refactoring, detection, code resolution, restructuring etc*

--***--

1. INTRODUCTION

Multi step software refactoring is an approach for restructuring an existing body of code and altering its internal structure without changing its external behavior by extract the design of original source code and explores the new design, in order to improve some of the nonfunctional attributes of the software. The code smell became complex to evaluate in programs. Code smells are usually not bugs, but it is not technically incorrect and prevents the program from functioning. Instead the code smell indicates weakness in design that may be slowing down in the development or increasing the risk of bugs or failure in the program [1]. So software engineers need detection tools and appropriate refactoring method for restructuring the programming codes by without alters the behavior of the original source code in the applications [1].

In this approach, we make use of detection tools for detection and resolution of code smells in java programs. The manual detection of code smells in huge system became complex and also consumes more time especially those involving more than one file or package. So, we go for the tools used are probable to detect code smells automatically and then multi step software refactoring is implemented in fully automated environment to remove code smells in the program [2]. As a result, the code smell in the programs are detected and refactored automatically.

2. RELATED WORK

In this Section, we first review related works addressing the code smells.

2.1 Refactoring in Automatically Generated Programs

Software systems experience incremental modifies more than time in arrange to deal with original necessities [1] [2]. Since the new design is not set for each new requirement in general, the addition of functionality brings the risk of degrading the quality of the design (structure) of the system. A common approach to mitigate this risk engages the use of refactoring. Refactoring aims at improving the design of accessible code by introducing structural alteration lacking changing its performance.

The motivation for refactoring the code of a system is that a well-designed system is normally easier to keep and expand. Refactoring is now a core element of software engineering put into practice, and is supported by the insertion of refactoring tools in well-used incorporated development environments [2]. Whereas the software refactoring is to obtain a quality software design and it decides the type of refactoring based on the situation. Newly, search-based approaches to mechanize the application of refactoring contain be proposed. These move toward cast the refactoring as an optimization trouble, where the objective is to get better the design excellence of a system based on a set of software metrics After originate the refactoring as an optimization problem by defining the solution representation, search operators and fitness function, several different methods can be applied to the problem of automated refactoring [3]. So far, the thought of habitual refactoring has been practical only to human-written code.

2.2. Other Related Techniques

Researchers have examined clone detection methods [6] to notice copy code in programs exceeding hundreds of thousands lines of code. All of these methods include known qualities and deficiencies, but as of today, little is recognized on where to fit these methods into the software preservation process. The clone detection methods contrast three delegate detection methods (simple line matching, parameterized matching, and metric fingerprints) by earnings of five small to medium cases and analyses the dissimilarity between the reported equal. Based on this trial, it shows that (1) Easy line matching is most excellent suited for a first crude impression of the copy code; (2) metric fingerprints work finest in blend with a refactoring tool that is able to remove copy subroutines; (3) Parameterized identical works optimum in mix together with extra fine-grained refactoring tools that work on the statement stage.

Detection methods of code clones are a two stage procedure which contains of a revolution and an assessment stage [4]. In the first phase, the source text is transformed into an internal format which allows the use of a more efficient comparison algorithm. Through the following assessment stage the definite matches are noticed. Due to its inner position, it is sensible to classify detection methods according to their inside format. An overview of the dissimilar methods available for each category while selecting a representative for each category like String Based, easy Line corresponding, Parameterized Line Matching, and Token Based.

3. DETECTION AND RESOLUTION OF CODE SMELLS

The detection tools which detect the different kinds of bad smells occur in the program, usually the detection tool recognizes the exact kind of code smell, e.g., the clone detection tool covers only clones in the program. The software engineer once confirmed the detected code smell and decides the software refactoring method to sanitary code smells. The resolution sequence of code smell may simplify the refactoring process, in such a way multi step software refactoring is carried out to evaluate the behavior of the program. This sequence of one type of code smell may affect the detection and resolution of another kind of code smell. Therefore, the code smells detection and resolution to make simpler by using the suitable detection and resolution sequences.

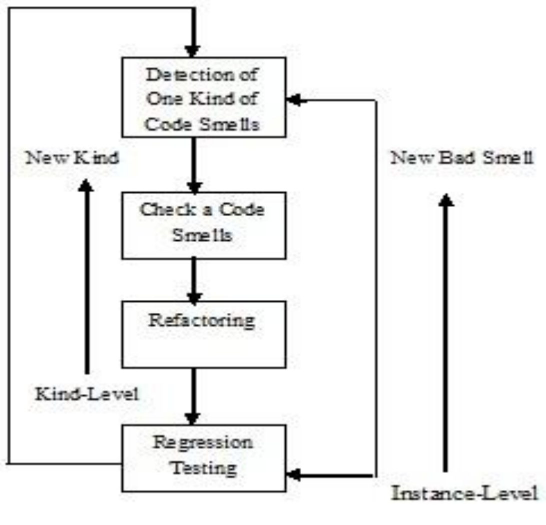

Fig -1: Detection and Resolution of Code Smell

As a result of this approach, we can automatically detect the evaluated code smells (Table 1) and also carried out an automated refactoring without a human interaction, by these results input of original java source code program contain code smells can be automatically refactored and finally getting an refactored source code without a code smells.

4. TOOLS AND METRICS

Software engineers necessitate tools to carry out the code detection and software refactoring either automatically or semi automatically [4]. In this paper, we make utilize of software tools, the integrated development environments (IDE), such as Eclipse, Microsoft Visual Studio and Intellij Idea JDEvAn and code-Imp support software refactoring. These tools are described in the following categories.

JDEvAn (Java Design Evaluation and Analysis) is an Eclipse plug-in developed in the University of Alberta. It evaluates a design evaluation history of software system and provides the information about the system history. In this paper, we use the java fact extractor and UMLDiff an design differencing algorithm. Code-Imp (Combinational Optimization for Design Improvement) is an automated framework for software refactoring developed in the University college of Dublin, Ireland. This automated framework is used to refactoring the code smells occurs in the java source programs. This tool support Java version 6 source codes as input and produces refactored source code.

Duplicated Code was detected by PMD and Long method, Large Class, Long parameter list were detected by using appropriate metrics like Cohesion metrics (LSCC, TCC, SCOM, CC), Coupling metrics (RFC, DCC, DAC, COF) and other metrics like (DIS, NOM, DAM, DSC). The metrics which included in the metrics suite is QMOOD, MOOD.

Table -1: Evaluated Code Smells

CODE SMELLS	RISK TYPE
Duplicated code	High
Long Method	High
Large Class	High
Long Parameter List	High

4.1 A Evaluated Code Smell

We focused four kinds of code smells at an initial stage. The evaluated code smells are listed in the Table 1. A short explanation of these code smells is obtainable here so that the paper can be understood on its own. We evaluate the relationship between these code smells and form the resolution sequence and we also recommend this resolution for commonly occurring code smells.

- **Duplicated Code:** The same code appears more than one location in a program is considered as a duplicated code.
- **Long Method:** The method which is harder to read or modify is consider as a long method. As a result long or complex method should be divided in to easy and well-named methods associates with the refactoring rules.
- **Large Class**: Large class consists of too many functions and tasks, which it making complex and confuse. To improve the understandability, the large class should be divided and assigns responsibilities in to simple ones.
- **Long Parameter List:** In programs, the methods which contain too many parameters are difficult to use and also difficult to alter. The parameters can be reduced by using simple objects.

5. RESOLUTION SEQUENCE OF CODE SMELLS

In this section, the recommended resolution sequence (Figure 2) is formed for the evaluated code smell. The directed graph is drawn based on the priority of the code smell and topological sorting algorithm is used to find the optimal sequence for the evaluated code smells. The topological sorting algorithm is desirable at the time of two or more vertices are available by using this algorithm randomly picks up the values which doesn't affect the original behavior of the system.

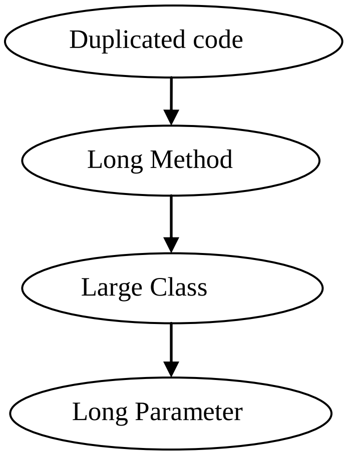

Fig -2: Resolution Sequence of Code Smell

6. MULTI STEP REFACTORING

The software refactoring is usually carried out in two ways. The First is refactoring in small step, in this method the refactoring is done with small size (scope for local files) and the second is refactoring is done systematically which attempt to refactor the whole system at an instance. In this paper we propose the multi step software refactoring.

We extract the original source code in to design level is presented in figure 3; it proposes new design model for the source code and explores the design. This design level improves software refactoring by initiates the multi step refactoring. The refactoring steps are: model evaluation, extract design, preferred design and refactoring. This multi-step software refactoring is resultant from the automated search based refactoring and it overcomes the problem in fully automated approach were by using an automated JDEv a tool in an Eclipse integrated environment.

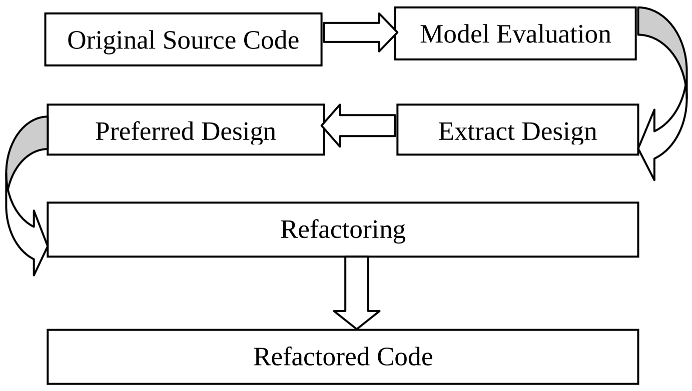

Fig -3: Multi Step Software Refactoring

The terms which we used in the multi step software refactoring is described for better understanding the concepts. Original source code consists of smells that is to be refactored

and the model evaluation step the model should be evaluated from the original source code. This model established according to the UML meta-model. Extract Design: The design extracted from the model evaluation, the design level may changes according to the change in the evaluation of the model.
The preferred design based on the UML model based on the original source code by explores a new design but updating the information based on the programmer needs. Refactoring: In this process the multi step refactoring is categorized in to detected and source level refactoring. The detected refactoring focus on design-level occurrences The source level refactoring proposes the refactoring directly on source code. Refactored Code: The refactoring is a process to clean up code smells in the original source code and getting the refactored code.

7. RESULTS AND DISCUSSION

Software refactoring can be carried out in multi-step automated refactoring approach and the results are presented in Table 2. Refactoring is carried out in Combinational Optimization for Design Improvement which supports Java 6 and several search types, it includes Java Source Metric to measure the metrics of Java source code. It is an Eclipse plug-in, detailed information of the simulation are presented and compared in Figure 4. From the figure, we observe that multi-step software refactoring approach has better results when compared to semi automated refactoring. Carrying out evaluation on these applications may help simplify the conclusions.

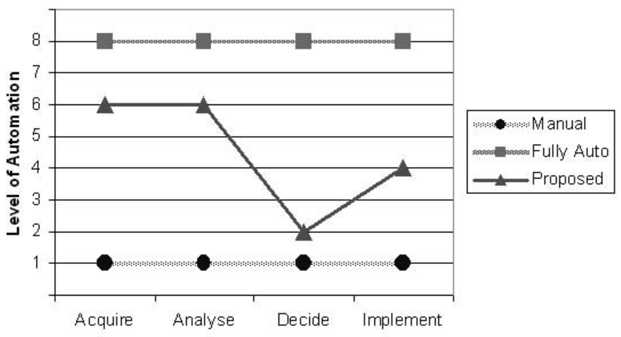

Fig -4: Automation levels for refactoring tools

The multi-step software refactoring shows the results of automated refactoring and the semi automated refactoring has taken undefined time to perform refactoring on detected code smell. The semi automated approach depends on man power and their performance but our proposed approach based on detection and refactoring tools. So, the evaluation results confirm the better performance on multi-step refactoring.

8. CONCLUSIONS AND FUTURE WORK

The different kinds of code smells are first annoyed and described the resolution sequence of various kinds of code smells, then proposes the final resolution sequence for commonly occurring code smells. Then we also illustrate the multi-step software refactoring for clean up the code smells automatically. These results the code smell can be detected and resolved automatically.

The main goal of our future work in this area is too carried out the automated refactoring for various kinds of code smells as we focused only four types. The rest of the remaining code smells are detected and refactored by using our multi step software refactoring approach or by using the fully automated framework code-imp for refactoring.

REFERENCES

[1] Hui Liu and Weizhong, “Schedule of Bad Smell Detection and Resolution: A New Way to Save Effort,”IEEE Trans. Software Eng., vol. 38, no. 1, pp. 220-235, Feb. 2012.
[2] Mens and T. Touwe, “A Survey of Software Refactoring,”IEEE Trans. Software Eng., vol. 30, no. 2, pp. 126-139, Feb. 2004.
[3] Moghadam and M. ´O Cinn´eide, “Code-Imp: a tool for automated search-based refactoring,” in Proceeding of the 4th workshop on Refactoring tools, ser. WRT ’11. New York, NY, USA: ACM, 2011, pp. 41–44.
[4] F.Tip, A.Kiezun, and D.Baeumer, “Refactoring for Generalization Using Type Constraints,” Proc. 18th Ann. Conf. Object- Oriented Programming Systems, Languages, and Applications, pp. 13- 26, Oct. 2003.
[5] Eclipse Foundation. Eclipse 3.4.2. http://www.eclipse.org/emft/projects/, 2011.
[6] Burd and J. Bailey, “Evaluating Clone Detection Tools for Use During Preventative Maintenance,” Proc. Second IEEE Int’l Workshop Source Code Analysis and Manipulation, pp. 36-43, Oct. 2002.
[7] Wettel and R. Marinescu, “Archeology of Code Duplication: Recovering Duplication Chains from Small Duplication Fragments,” Proc. Seventh Int’l Symp. Symbolic and Numeric Algorithms for Scientific Computing, p. 63, 2005.

BIOGRAPHIES

M. Lakshmanan received the M.E degree in Software Engineering from Anna University in 2013. He is an assistant professor in the CSE department at the Gnanamani College of Engineering. His current research interests include software refactoring, design pattern, and software evolution.

S. Manikandan, P.G Scholar, department of Computer Science and Engineering at the Gnanamani College of Engineering He is particularly interested in software engineering, testing, object-oriented technologies and software reuse.