

A Quantitative Model for Improving the Effectiveness of the Software Development Process using Refactoring

K.Usha¹, N.Poonguzhali², E.Kavitha³

¹Lecturer, MIT, Pudhucherry.ushavaratharajan@gmail.com,India

²Assistant Professor, MIT, Pudhucherry, India, poongulee@yahoo.co.in

³Senior Lecturer, V.R.S College of Engineering, T.N India, e_kavi_2000@yahoo.com

Abstract - Software development is a mentally complicated task. Different software development methodologies and quality assurance methods are used in order to attain high quality, reliable, and bug free software. eXtreme Programming (XP) is a software development discipline in the family of agile methodologies that contributes towards quality improvement using dozen practices. One of the important practices in XP is Refactoring. Refactoring which can be defined as “a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior”. Although there has been a large amount of research investigations for this concept, but there has been little work done in quantitative approach. The main objective of the research is to develop a software development model using Refactoring practices. This work also shows the evaluation of effectiveness of the proposed software development model.

Keywords – refactoring; software development; eXtreme programming (XP)

I. INTRODUCTION

Software Refactoring is a technique to enhance the maintainability of software, improve reusability and understandability of the software. Although the refactoring concept itself is considered to be effective, there are few quantitative evaluation of its impact to the software maintainability, reusability and understandability. We propose a quantitative evaluation method to measure the effect of refactoring in terms of maintainability, reusability and understandability. We focused on the metrics to evaluate the refactoring effect. By comparing the metric values before and after the refactoring, we could evaluate the effect of refactoring. We applied our method to a three different software projects and shown that our method was really effective to quantify the refactoring effect and helped us to choose appropriate refactorings.

Refactoring is behavior-preserving source-to-source program transformation process [1]. Refactoring is basically the object-oriented variant of restructuring: “the process of changing a [object-oriented] software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure” [2]. In the refactoring process, a change were done to the system leaving its

behavior unchanged, but enhances some non-functional quality like simplicity, flexibility, understandability, etc., [3].

II. THE DEVELOPMENT MODEL

Software evolution can be time-consuming and tedious, and often accounts for up to 75% of costs associated with the development of software-intensive systems. Lehman’s laws of software evolution [4], [5] state that while the functionality of a system increases over time, there is a corresponding decrease in quality and increase in internal complexity. Refactoring is a process that helps mitigate the problems of evolution by addressing the concerns of internal complexity and quality.

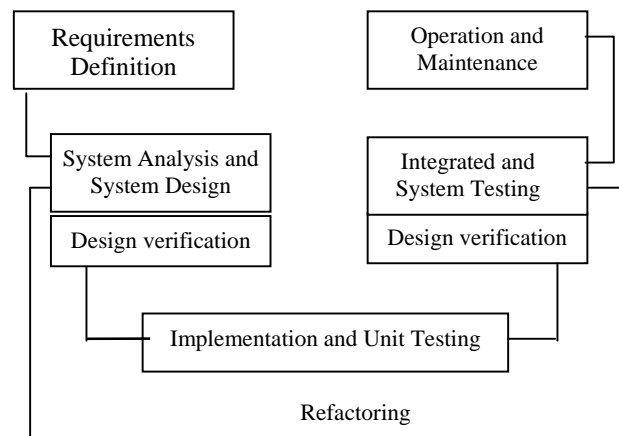


Figure 1. Software Development Process using Refactoring

In this paper, we discuss our experiences with three different projects that followed a refactorings-based development. We review the major milestones in their development and the activities between them, we discuss some interesting refactorings that were applied between versions, and we report on several metrics we have collected for these versions. We conclude with some reflections on the trends we have noticed in those three projects.

III. REFACTORING PROCESS

we constructed a systematic refactoring process which helps us to apply appropriate refactorings to the target system. The program refactoring process consists of three major phases: Identification Phase, Proposal Phase, Evaluation Phase, and Application Phase

A. Identification Phase

In this phase, the area in which the source code to be refactored is identified. This is done by means of code smell detection. The figure indicates the operations performed in the Identification Phase.

- Code Smell Detection: “Code-smell” is defined as a program characteristic which alludes to the necessity of program refactoring [Fowler]. For instance, “duplicated code” is one type of bad-smell.
- Code smell analysis: “Code-smell” does not necessarily lead to a single particular refactoring. Generally speaking, duplicated codes are to be unified. It is preferable, that such a unified code is implemented in a parent class rather than other arbitrary places if the duplication is only found among sibling subclasses. By analyzing bad-smells precisely, we can identify a better solution.

B. Proposal Phase

- Refactoring Planning: After analyzing various bad-smells, a number of refactoring candidates would be identified. Some will be easy to perform and some others will be hard to realize. Some will provide great improvement while some others will contribute little to maintainability. Moreover one candidate could be a counterpart of another so those two refactorings cannot be applied at the same time. For instance, “Extract Method” turns a code fragment which can be grouped together into a method whose name explains the purpose of the method.
- Improvement Planning: The objective of our selection was to investigate those refactorings which redistribute responsibilities either within the class or in between classes. This redistribution is the main principle to address coupling and cohesion problems. Along with the refactoring for detected code smell, more refactorings are investigated to improve coupling and cohesion of the code.
- Proposal of Refactoring Candidates: The output of this phase will be the list of refactoring candidates that can be applied in the original source code to get refactored source code.

C. Evaluation Phase

The main objectives of the refactoring practice in the software development process are the enhancing the

maintainability, reusability and improving the clarity of the code.

Evaluation phase can be implemented in two modules.

- Selection of appropriate quantification metrics for maintainability, reusability and understandability.
- Measuring and comparing metrics before and after refactoring

IV. CASE STUDY

To evaluate the proposed theory, we implemented the following experiments. We choose three software projects. The projects are

- “Inventory Application” (program 1)
- “Tic Tac Toe Game” (program 2)
- “Binary Search Tree” (program 3)

We applied Refactoring Assistant to those projects and found out many bad smells in the code. We identified the program weak points in terms of coupling and cohesion. With all this, we singled out the problems and proposed refactoring candidates for those problems. The proposed evaluation model is used to evaluate the effect of each refactoring candidates in terms of maintainability, reusability and understandability. Evaluation Model helps us to choose appropriate refactorings to be applied.

A. Graphical Results

The graphical representations of the comparison metric values obtained for different programs.. The following graphical figure 2 shows that the Cyclomatic complexity values of refactored programs found to be less than that of the original code.

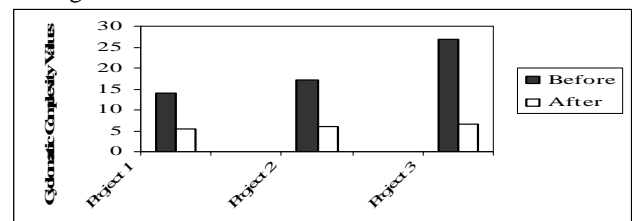


Figure 2: Comparison of Cyclomatic Complexity Metric for different Programs

The following graphical figure 3 shows that the coupling values of refactored programs found to be less than that of the original code.

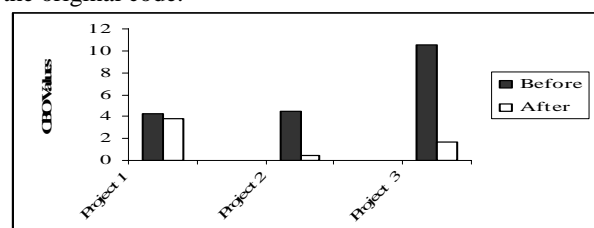


Figure 3: Comparison of CBO Metric for different Programs

The following graphical figure 4 shows that the cohesion values of refactored programs found to be less than that of the original code.

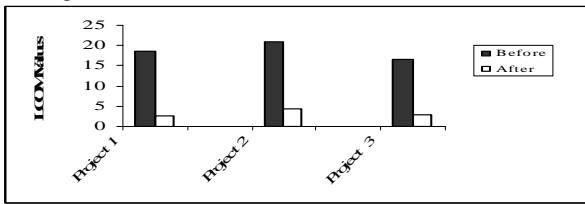


Figure 4: Comparison of LCOM Metric for different Programs

The following graphical figures 6 and 7 show that the MHF and AHF values of refactored programs found to be more than that of the original code.

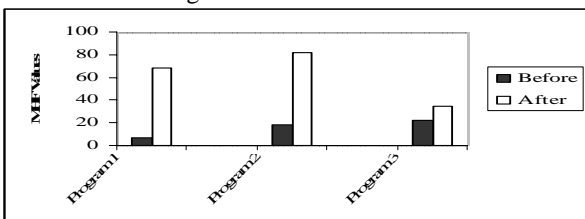


Figure 6: Comparison of MHF Metric for different Programs

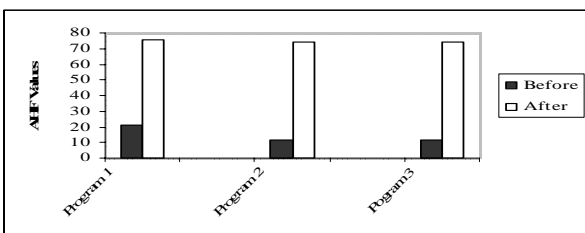


Figure 7: Comparison of AHF Metric for different Programs

B. Result Analysis

Refactoring leads to

- Decrease in Average Number of Attributes. This makes the code simple and less complex
- Decrease in Average Number of Methods. This means that a class is not overloaded with more functionality. This makes the code simple.
- Increase in Number of classes. This comprehensibility of the system increases, which makes the system easier to maintain.
- Decrease in the average Cyclomatic complexity. The system complexity reduces.
- Decrease in the value of CBO. This decreases the dependencies exists between classes, which improves reusability
- Maintains Moderate values for CF. this enhances maintainability and improves reusability without any side effects i.e., excessive code duplication.

- Decrease in the value of LCOM. This results in increases cohesion values, thereby improving reusability and Understandability of the code.
- Improvement in the value of MHF, AHF. This process improves the tasks of modifiability

V. CONCLUSION

Refactoring has been reported to have benefits in software development process. In this work we proposed a model for software development using refactoring. We conducted experiments for refactoring process. In this research work we propose a refactoring methodology consisting of three phases, Identification phase, Proposal and Application Phase, Evaluation Phase. The Evaluation parameters are taken as Maintainability, Reusability and Understandability. we can conclude that, refactoring activity in the software development process leads to improvement in reusability and Understandability of the code, thereby enhancing maintainability of the code.

VI. FUTURE ENHANCEMENTS

.In terms of future work, we will look into the possibility of a developer opinion based study of refactorings and code smells; in other words, which refactorings do developers prefer doing (if any) and also what code smells do they prefer to eradicate. We would also like to investigate the link between testing, refactoring and the incidence of faults found in the code

REFERENCES

- [1] Beck, K., "Extreme Programming Explained: Embrace Change." 2000, Reading, Mass: Addison-Wesley.
- [2] Tom Mens, Tom Tourw', "A Survey of Software Refactoring", IEEE Transactions On Software Engineering, Vol. Xx, No. Y, Month 2004
- [3] Frank Simon, Frank Steinbruckner, Claus Lewerentz, "Metrics Based Refactoring", 2001 IEEE
- [4] John T. Nosek, "The Case for Collaborative Programming", Communications of the ACM. March 1998/Vol. 41, No. 3.
- [5] Charlie McDowell, Linda Werner, Heather E. Bullock, and Julian Fernald, "Pair Programming Improves Student Retention, Confidence and Program quality", August 2006/Vol. 49, No. 8 Communications of the ACM.
- [6] S. M. Henry and D. G. Kafura, "The evaluation of software systems' structure using quantitative software metrics", Software - Practice and Experience, 14(6):561-573, 1984.