Identificaton and Evaluation of Code Using Refactoring Method

Amalan S A , Yuvaraj N
*1 *Assistant Professor*
Excel Engineering College,Komarapalayam,Tamil Nadu.India

*2 *Assistant Professor*
Excel Engineering College,Komarapalayam,Tamil Nadu.India

***Abstract*— Software testing is a critical element of software quality assurance that represents the ultimate review of specifications, design and coding. In computer programming, code smell is the symptom in the source code of a program indicating a deeper problem. Code smells are usually not bugs, they are not technically incorrect and don't currently prevent the program from functioning. Instead, they indicate weaknesses in design that may be slowing down development or increasing the risk of bugs or failures in the future. Code and design smells are poor solutions to recurring implementation and design problems. Bad smells are signs of potential problems in code. Detecting and resolving bad smells remain time-consuming for software engineers .Numerous bad smells have been recognized, the sequences in which the detection and resolution of different kinds of bad smells are performed rarely because software engineers do not know how to optimize sequences or determine the benefits of an optimal sequence. So, a new sequence for different kinds of bad smells has been implemented, to simplify the detection and resolution of bad smells based on refactoring method. This system reduces the code complexity occurred in coding environment and improves the quality of software.**

***Keywords*— Scheme, bad smell, software refactoring, effort, detection, schedule.**

I. INTRODUCTION

Data refactoring is the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure. It includes improvement of code readability and reduced complexity to improve the maintainability of the source code, as well as a more expressive internal architecture or object model to improve extensibility. Refactoring aim to reverse this decline in software quality by applying a series of small, behaviour preserving transformations each of which improves a certain aspect of the system. Software systems have to be flexible in order to evolving requirements.

Introduction about refactoring is defined by the benefits, problems and also guidelines for applying refactoring methods when to refactor, which techniques to use, and how to apply them, and when to stop. Java and some refactoring methods are used for modifying the techniques in Delphi. Most software spends far longer in maintenance more than 90% of the program lifetime. Maintenance means fixing bugs, changing the program behavior to meet changing requirements, and adding new features. All of these activities mean modifying or extending existing code. So the readability and maintainability of the code base is the paramount features of any program development.

Before refactoring a section of code, a solid set of automatic unit tests is needed. The tests should demonstrate in a few seconds that the behavior of the module is correct. The process is then an iterative cycle of making a small program transformation, testing it to ensure correctness, and making another small transformation. If at any point a test fails, the last small change is undone and repeated in a different way. Through many small steps the program moves from where it was to where want it to be. Proponents of extreme programming and other agile methodologies describe this activity as an integral part of the software development cycle. code is easy to read and the intent of its author is easy to grasp. This might be achieved by reducing large monolithic routines into a set of individually concise, well-named, single-purpose methods. It might be achieved by moving a method to a more appropriate class, or by removing misleading comments. Extensibility is easier to extend the capabilities of the application if it uses recognizable design patterns, and it provides some flexibility where none before may have existed.

II. BACKGROUND

Up to 75% of the costs associated with the development of software systems occur post-deployment during maintenance and evolution. Software refactoring is a

process which can significantly reduce the costs associated with software evolution. Refactoring is defined as internal modification of source code to improve system quality, without change to observable behaviour. Tool support for software refactoring attempts to further reduce evolution costs by automating manual, error-prone and tedious tasks. Although the process of refactoring is well-defined, tools supporting refactoring do not support the full process. Existing tools suffer from issues associated with the level of automation; the stages of the refactoring process supported or automated the subset of refactoring that can be applied, and complexity of the supported refactoring. The work offers the following contributions to resolve the above problem: Relationships are analysed among different kinds of bad smells and their influence on detection and resolution sequences. The need to arrange detection and resolution sequences of different kinds of bad smells using a motivating example are also identified. A resolution sequence for commonly occurring bad smells are recommended.

III. METHODOLOGY

The framework consists of detecting the clone, evaluating the bad smell, resolution sequence of bad smell using refactoring method .

A. Detecting the clone

In this module for detecting bad smells equipped with bad smell detection tools and automatic or semiautomatic refactoring method for cleaning up bad smells. First develop a detection tool to identify a specific type of bad smell is clone(a detection tool usually uncovers only one kind of bad smell, e.g., clone detection tools is insensitive to bad smells other than clones). The detection tool proposes initial results that require manual confirmation. Once the detected bad smell is confirmed, the software engineer decides how to refactor it. Selected refactoring rules are manually or semi-automatically applied to the bad smells with the help of refactoring tools. Then, the software engineer moves on to the next kind of bad smells detection and repeats the process until all kinds of bad smells have been detected and resolved. As a result, different kinds of bad smells are detected and resolved one after the other. Using kind level scheme to detect one after another bad smells.

B. Evaluating the bad smells

In this module evaluating taken for the following bad smell,

Long Method

The longer a method is, the harder it is to read or modify. Consequently, a long and complex method is divided into short and well-named methods with refactoring rules e.g. extract Method. As a solution to Long Method, some parts of the method may be extracted as new methods. Usually the extracted new methods are called within the old one in the original location; thus, the extraction does not shorten the parameter list.

Large Class

Large classes usually try to take too many responsibilities, making them complex and confusing. To improve their readability and maintainability, large classes is divided into smaller ones, each for a single responsibility. To eliminate Large Class, large classes are decomposed into a few small ones On the other hand, long methods are decomposed into a few small methods to dispel Long Method. Consequently, resolving Large Class and Long Method leads to redistribution of responsibilities at class and method levels, respectively. Carrying out the redistribution of responsibilities from the bottom up makes for reasonable and thorough redistribution.

C. Resolution sequence of bad smells using refactoring method

In this module develop the graph to resolve the bad smells here assume all bad smells of the same kind would be detected and resolved before the next kind of bad smell is detected.

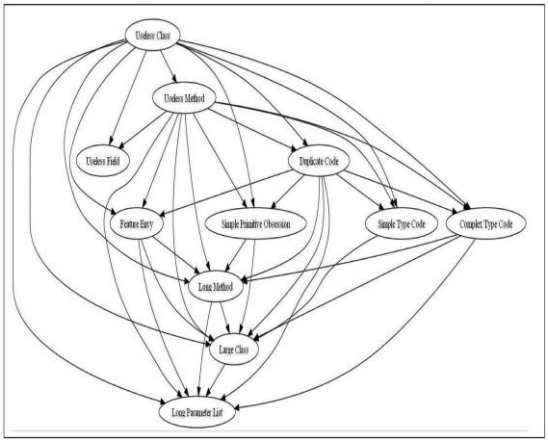

Fig 2.1**: Sequence of bad smells**

The sequence of code smells where related with each other and has combined with one another. It has specific orrelation among each test smells. Based on the above sequence the test smell is detection and then the code is under resolution based on the refactoring method.The refactoring method ignore the duplication code and useless code in the application.

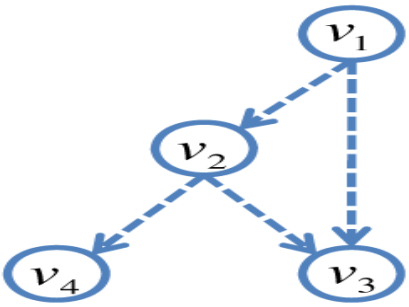

Fig 2.3: Redundant edge

Remove more number of vertices for one single feature can removed by the following algorithm propose a new algorithm for the final resolution sequences removing redundant edges from Fig .For convenience, define the following symbols p(v1; v2) A path from vertex v1 to vertex v2 containing more than one edge (v1; v3) A direct edge from vertex v1 to vertex v3.

An edge e (v1; v3) is redundant if and only if there is another path p(v1; v3) in parallel to e(v1; v3). Removing edge e(v1; v3) would not change the topological order of the vertices, but removing any other edge from Fig. result in different topological order. If the algorithm is applied to the graph in Fig, the algorithm would remove e (v1; v3) because the path p(v1; v2; v3) is parallel to e(v1; v3). Other edges would be retained because no path is longer than 1 in parallel to e(v1; v2), e(v2; v3), or e(v2; v4).

Using this refactoring method the coding has been changed and the complexity is reduced in the coding, execution time consuming has been reduced. Then code lead the software to de standard and also improve the code quality based on this refactoring method.

IV. PERFORMANCE EVALUATION

This section describes the methodology results for detecting and resolution for bad smells.

INPUT

In this system, the process of detecting and resolving the bad smells from the coding environment. So some sample java program is taken and resolved through the refactoring method.

OUTPUT

Then the clone is detected from the given sample program and the coding detail is evaluated. The result is given below .Using search based refactoring new features like coupling also detected and then refactored to minimize the class function and also reduce the code complexity.

Code is reduced then the execution and performance level also increase .Then, the software quality is improved and metrics is calculated.

Table 1: Detecting the clone

APPLICATIONS	Value for Cyclomatic Complexity	
	Before Refactoring	After Refactoring
1.Calculator-APP a. Sum b. Sub	18	16
2.Banking – APP a. Transaction b. Withdraw	24	16
3.Employee Detail a. Insert Values b.View Table	20	15
4.Game –APP a.Puzzle	14	10

Table 4.1: Evaluating the bad smells

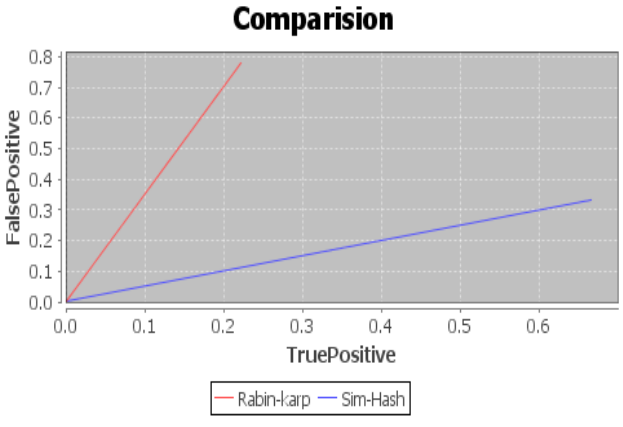

Fig 4.2:Comparison About Before And After Refactoring

V. CONCLUSIONS

In this project a new sequence has been implemented based upon bad smells are detection. The evaluation of bad smells are resolution based on the refactoring method by using integrated development environment.

The contribution of the paper, to reduce the code complexity, produce software quality and reduce the time consuming for execution of the code. Some of the effort range also measured and calculated based upon this new sequence of bad smells. So the complexity of the code is reduced and execution of the code has time consuming.

References

[1] Hui Liu and Weizhong Shao, "Schedule of Badsmell Detection and Resolution:A New Way to Save Effort," IEEE Trans. Software Eng., vol. 38, no. 1 Jan/Feb. 2012.
[2] W.F. Opdyke, "Refactoring Object-Oriented Frameworks," PhD
dissertation, Univ. of Illinois at Urbana-Champaign, 1992.
[3] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, Refactoring: Improving the Design of Existing Code.Addison Wesley Professional, 1999.
[4] W.C. Wake,Refactoring Workbook.Addison Wesley, Aug. 2003.
[5] http://wiki.java.net/bin/view/People/SmellsToRefactorings, 2011.
[6] W.G. Griswold and D. Notkin, "Automated Assistance for Program Restructuring,"ACM Trans. Software Eng. and Methodol-ogy,vol. 2, no. 3, pp. 228-269, July 1993.
[7] F. Tip, A. Kiezun, and D. Baeumer, "Refactoring for General-ization Using Type Constraints," Proc. 18th Ann. Conf. Object-Oriented Programming Systems, Languages, and Applications,pp. 13-26, Oct. 2003.
[8] T. Mens, N.V. Eetvelde, and S. Demeyer, "Formalizing Refactorings with Graph Transformations," J. Software Maintenance and Evolution: Research and Practice, vol. 17, no. 4, pp. 247-276, 2005.
[9] R. Koschke, "Identifying and Removing Software Clones,"Soft-ware Evolution,T. Mens and S. Demeyer, eds., pp. 15-36, Springer, 2008.
[10] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A Multi-Linguistic Token Based Code Clone Detection System for Large Scale Source Code," IEEE Trans. Software Eng., vol. 28, no. 7, pp. 654-670, July 2002.
[11] S. Ducasse, M. Rieger, and S. Demeyer, "A Language Independent Approach for Detecting Duplicated Code,"Proc. Int'l Conf. Software Maintenance,pp. 109-118, 1999.
[12] E. Burd and J. Bailey, "Evaluating Clone Detection Tools for Use During Preventative Maintenance,"Proc. Second IEEE Int'l
Work-shop Source Code Analysis and Manipulation,pp. 36-43, Oct. 2002.
[13] B.S. Baker, "On Finding Duplication and Near-Duplication in Large Software Systems,"Proc. Second IEEE Working Conf. Reverse Eng.,pp. 86-95, July 1995.
[14] R. Wettel and R. Marinescu, "Archeology of Code Duplication: Recovering Duplication Chains from Small Duplication Frag-ments,"Proc. Seventh Int'l Symp. Symbolic and Numeric Algorithms

[15] Eclipse Foundation. Eclipse 3.4.2. http://www.eclipse.org/emft/ projects/, 2011.
[16] Microsoft Corporation. Microsoft Visual Studio 2008. http:// www.microsoft.com/, 2011.
[17] JetBrains Company. IntelliJ IDEA 8. http://www.jetbrains.com/ idea/, 2011.
[18] E. Mealy and P. Strooper, "Evaluating Software Refactoring tool Support,"Proc. Australian Software Eng. Conf.,pp. 331-340, 2006.
[19] E. Murphy-Hill and A.P. Black, "Refactoring Tools: Fitness for
Purpose,"IEEE Software,vol. 25, no. 5, pp. 38-44, Sept./Oct. 2008. [20] E. Mealy, D. Carrington, P. Strooper, and P. Wyeth, "Improving
Usability of Software Refactoring Tools,"Proc. Australian Software Eng. Conf.,pp. 307-318, Apr. 2007.
[21] S. Bouktif, G. Antoniol, E. Merlo, and M. Neteler, "A Novel Approach to Optimize Clone Refactoring Activity,"Proc. Eighth Ann. Conf. Genetic and Evolutionary Computation,pp. 1885-1892, July 2006.

[10] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A Multi-Linguistic Token Based Code Clone Detection System for Large Scale Source Code," IEEE Trans. Software Eng., vol. 28, no. 7, pp. 654-670, July 2002.
[11] S. Ducasse, M. Rieger, and S. Demeyer, "A Language Independent Approach for Detecting Duplicated Code,"Proc. Int'l Conf. Software Maintenance,pp. 109-118, 1999.
[12] E. Burd and J. Bailey, "Evaluating Clone Detection Tools for Use During Preventative Maintenance,"Proc. Second IEEE Int'l
Work-shop Source Code Analysis and Manipulation,pp. 36-43, Oct. 2002.
[13] B.S. Baker, "On Finding Duplication and Near-Duplication in Large Software Systems,"Proc. Second IEEE Working Conf. Reverse Eng.,pp. 86-95, July 1995.
[14] R. Wettel and R. Marinescu, "Archeology of Code Duplication: Recovering Duplication Chains from Small Duplication Frag-ments,"Proc. Seventh Int'l Symp. Symbolic and Numeric Algorithms
for Scientific Computing, p. 63, 2005.
[15] Eclipse Foundation. Eclipse 3.4.2. http://www.eclipse.org/emft/ projects/, 2011.
[16] Microsoft Corporation. Microsoft Visual Studio 2008. http:// www.microsoft.com/, 2011.
[17] JetBrains Company. IntelliJ IDEA 8. http://www.jetbrains.com/ idea/, 2011.