International Journal of Emerging Technology and Advanced Engineering
Website: www.ijetae.com (ISSN 2250-2459 (Online), An ISO 9001:2008 Certified Journal, Volume 3, Special Issue 1, January 2013)

International Conference on Information Systems and Computing (ICISC-2013), INDIA.

SOFTWARE REFACTORING WITH APPROPRIATE RESOLUTION ORDER OF CODE SMELLS

B.Maheswari[1], T. Subashini[2]

[1]*Student*, [2]*Assistant Professor, Department of CSE, Velammal engineering college, Anna University, India*

Email: subhatanish@gmail.com / maheswarivec@gmail.com

Abstract

Code smells are structured characteristics of software that may indicate a code or design problem.It makes a software hard to evolve and maintain and may trigger refactoring of code.It have a different types of code smells,refactoring tools,but it doesn't know the sequence of code smells to be resolved first.Identifying and detecting the code smells are performed rarely because all types in the code smells are still not yet performed. Refactoring tools can be optimize the code smell detection. Here they proposed the all types in code smells are identified by using the specific refactoring tool. This paper reduces the code detection ranging from 20-30 percent than the existing approach.

Indexterms-- **software refactoring,schedule,code smell detection**

I. INTRODUCTION

Refactoring is a method which used to rearrange and modify the existing code in a way that the intenional behaviour of code stays the same[1].Refactoring allows to streamline,simplify and improve both performance and readability of your code[1].one of the key issue in software refactoring is source code that should be refactored. It should be implemented by the object oriented programs. Kent beck and Martin fowler call them Bad smells[1], indicating that some part of the source code are terrible. Sometimes in other words bad smells are assigned as the duplicate code. Duplicate code is a computer program sequence of source code that occurs more than once. Improving the design that often includes removing duplicate code.

Identifying and detecting these types of bad smells in code are performed individually or automatically. Software engineers do not know the sequence of occurring different types of bad smells in code. Existing system approaches the all different types of bad smells cannot be identified by the refactoring tools. While detecting the code it cannot performed all the types are evaluated. Few of the files can be identified in the existing system.

In proposed system, first detecting in the large scale system, that should be performed by manually. Second, bad smells are detected automatically or semi automatically by the use detection tools. Third all the types can be implemented by the detection tool and refactor the bad smells in code.

In this paper focused on the different kind of bad smells are identified by the specific refactoring tools. It illustrates how to arrange the sequence for commonly occurring bad smells in code.

Section 2 describes software refactoring activities can be used in this paper, section 3 describes the detection of tools to identify the code smells, section 4 describes the conclusion.

II. SOFTWARE REFACTORING

Erica Gylmm et al and Paul strooper et al [1] suggest in that refactoring can reduce the cost of the software evaluation. This paper deals with the evaluating software tool support based on the DESMET Feature analysis technique. The current level is described and implemented without automating the detection for the application. DESMET involved in industry driven assessment of software tools and methods .Existing system supports for refactoring is inadequate due to incompleteness of the entire process.Top level categories for this feature set is:

1. Supplier assessment	2. Maturity of tools
3. Economic issues	4. Ease of introduction
5. Realibility	6. Efficiency
7. Usability	8. Compatibility
9. Refactoring specific.	

Tom Mens [2] suggest that overview of existing research in the field of refactoring. This paper provides different set of techniques and mechanisms are used in the different criteria.

This technique is needed to reduce the software complexity and improving the software quality.

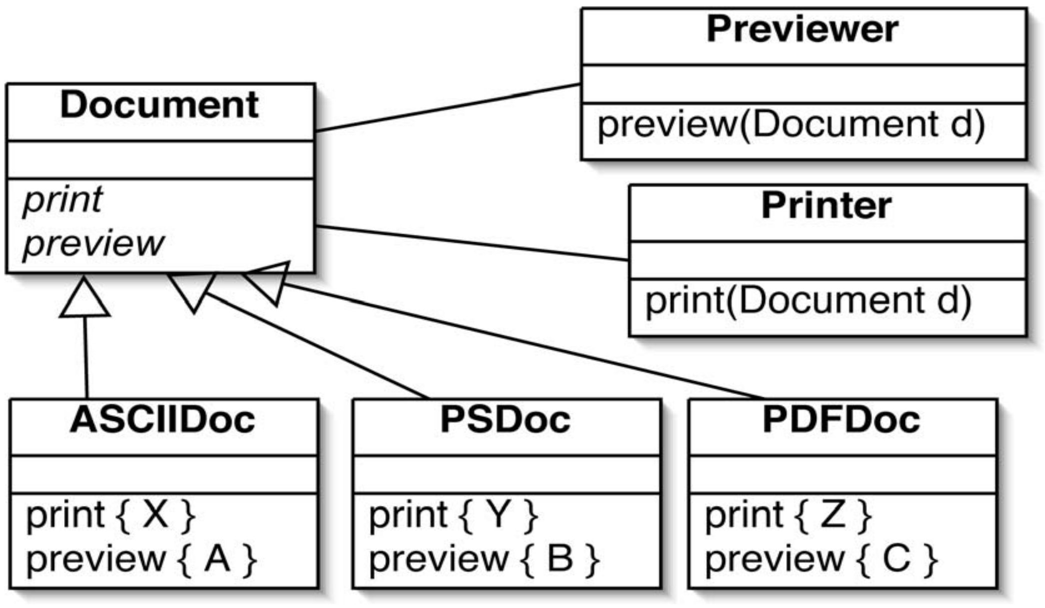

Fig 1.Document class hierarchy and helper classes

In fig 1 *Document* class are distributed over the classes.In *Document* Class it is need to change the every subclass in the document using the *Helper* Class.*Document* class doesn't have the no relationship between the *helper* classes. To overcome this problem, the refactoring techniques is used and add a new functionality in a new sub class called *visitor*. Different types of refactoring are used in the*visitor*.1.MoveMethod refactoring 2. RenameMethod refactoring 3. AddClass refactoring 670] 4. Add Method refactoring 5. Extract Methodre factoring 6. Pull Up Method refactoring

A variety of formalisms and techniques have been proposed to deal with the one or more refactorings. The use of assertions (pre conditions, post conditions, and invariants) and the use of graph transformation. Software Reengineering and Agile software development are performed in this paper. Reengineering can be involved in the modifying a legacy system.

Extreme programming(XP) is a main component, suggests supporting a continuous reengineering. Van Deursen et al suggests the distinct set of code smells and involves the test-specific refactorings.

Erica Mealy et al [3] suggest that the improving refactoring tools can reduce the cost. Usability of the software refactoring is used in the ISO 9241-11.During refactoring, user must analyse the large collection of data. In this paper they developed 81 usability requirements for the refactoring tools. In existing environment: Ecilpse 3.2, RefactorIT 2.5.1, condenser 1.05, and Ecilpse 3.2 with the Simian UI 2.2.12 plugin.

The major goal is to improve the software quality without changing the observable behaviour. Software refactoring tools developed by the theory & practices. Refactoring can be broken down into three tasks:

1. Detection of code smells
2. Transformation to remove the code smell
3. selecting the refactoring transformation.

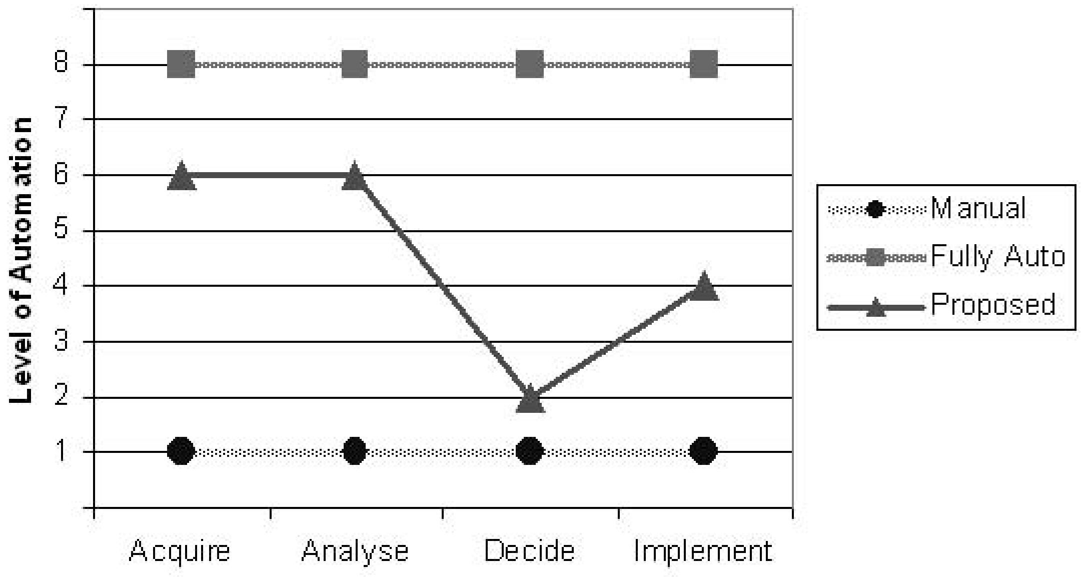

Fig.2 Automation levels for refactoring tools

The Fitts' MABA-MABA List describes the law of task allocation to identifies by the men or by the machines. Inductive reasoning and deductive reasoning are general facts in the principle. Code smell can be identified by the deductive reasoning it should be fully automated. It have the guidelines are Consistency, Errors, Information Processing, User Experience, Design for the user(s),user control, Goal Assesment, and Ease of use. This Paper derives from the complete guidelines for the usability and four refactoring tools can be evaluated. The future work is implemented by the Semi-automated approach to improve the quality.

III. Detection Of Code Smells

Nikolaos Tsantalis et al [4] was proposed to identifying and removing the type-checking bad smell with appropriate in java source code using the JDeodarant prototype.

Type-checking code is introduced by the variation of algorithm, it should depending on the value of attribute. Two methodologies are used here, first one is identification of type-checking bad smells focused on the switch/if statement should contain more than one case considered to be a valid. second one is replace type code with state/strategy refactoring and replace conditional with polymorphism refactoring. Here the further challenges are used to perform in the large scale system to assess the precise and recall of the tool.

MoveMethod refactoring [5] was proposed to identify the feature envy bad smells.The algorithm can be used to identify the set of entities and classes can extracts the all behaviour of conditions. Coupling and cohension problems are enhanced by the Feature envy bad smells.MoveMethod refactoring is the only solutions for Feature envy design problem.The proposed methodology describe the four parts to evaluate the process.

TABLE 1
Evaluated Bad Smells

	Bad Smells
1	Duplicated Code
2	Long Method
3	Large Class
4	Long Parameter List
5	Feature Envy
6	Primitive Obsession
	6.1: Simple Primitive Obsession
	6.2: Simple Type Code
	6.3: Complex Type Code
7	Useless Field
8	Useless Method
9	Useless Class

The first one follows the qualitative analysis and second should be evaluated by the coupling and cohension process. Third and fourth involves the refactoring suggestions, time required for extracting the refactoring. Here the future work is considering the two open source projects and to improve the design of the quality by the use of Move Method refactoring.

Eva van emden et al [6] was proposed a technique to improve a software quality using software inspection. This paper was focused on the code browser for detecting and visualizing code smells .One of the advantage in software inspection is tested after the software should be analyzed. Most of the problems are identified in the life cycle starts from the beginning. Code smell browser can have the two phases 1.code smell extraction 2.visualization. Extraction supports the parsers, syntax directed editors and tools such as interpreters, type checkers and source to source transformations. During the extraction first source code is parsed and source model is generated that describes program structure and code smells. The advantages is when the source code is do not compile by itself because of incompleteness. It can be expressed by the using of Java programs. Visualization can be detected by the jCosmo tool. It can provide the customized views of extracted data and special filtering can be used to show or hide the nodes

Salah Bouktif et al [7] proposed to carried out the testing community. It contribute the work by using the mutation and coverage criteria based testing.

It focused on the mutation testing for java code and coverage based testing for c code. This paper proposes to schedule quality improvement and constraints using the GRASS open source software.1000 duplicate code are identified in the GRASS. The main contribution of this paper to solve quality improvement with the use of Multi-constrained Knapsack problem. Proposed system involves a novel model to estimate the clone refactoring.

Richard Wettel et al [8] suggest that the code duplication is a common problem and well-known sign of bad design. However, duplicated fragments rarely remain identical after they are copied; they are often times modified here and there. By recovering such duplication chains,the maintenance engineer is provided with additional cases of duplication that can lead to relevant refactorings, and which are usually missed by other detection methods. we propose an approach that merges such small fragments that belong together, thus providing the maintainer with some additional duplication blocks, otherwise granted with less importance or not detected at all (due to filtering).

This paper was proposed by the archelogy metaphor context. The scatter-plot approach was successfully applied in the code duplication detection field starting with the early ’90s [1], [4], [5]. Duplication Chain can be a complex element, composed of a number of smaller exact clones,and separated by non-matching gaps.

An exact chunk is a non-altered part of a duplicated block, or in the context of the archeology metaphor an artifact found in its place. A *non-matching gap* reflects the changes that have been made to the originating duplicated block, in terms of lines of code (insertion, deletion, modification Two closely related characteristics of a duplication chain, directly influenced by the adaptation process are the *type* and the *signature*. The *type* of the duplication chain provides a summary of the adaptations made to the duplication block. The *signature* further extends the meaning of the type by capturing the structural configuration in terms of exact chunks, non-matching gaps and the metrics around them. This paper follows the three phases

1. Code Preprocessing
2. Populate the scatter plot
3. Build the duplication chain.

The duplicated code for the selected chain is presented in highlight in the code visualization panel, an important feature for the process of manual validation of the results. One of the major advantages of the approach presented in this paper is that it provides additional duplications to the ones detectable by other traditional methods. In extension to these desiderates, we would be interested in researching on the information we could extract from the signatures of code duplication chains and to provide assistance in the refactoring process, based on identified patterns.

IV. Conclusion

In this surveys we discussed about the different types of code smell can be identified by the use of specific refactoring tools. Existing approach they used in extreme programming floss refactoring but it can identifies only a few local files.

In proposed methodology batch model process can be implemented. It can be used for the large system thoroughly in one attempt. Then, illustrated how to arrange such a resolution sequence for commonly occurring bad smells. It can also carried out evaluations on two nontrivial applications to validate the research.

REFERENCES

[1] E.Mealy and P.Strooper,"Evaluating software Refactoring Tool support,"Proc.Australian Software Eng.Conf.,pp. 331-340,2006.

[2] T.Mens and T.Touwe, "A survey of software refactoring" IEEE Trans.software Eng.,vol.30, no.2, pp. 126-139, Feb 2004.

[3] 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.

[4] N.Tsantalis,T.Chaikalis,and A.Chatzigeorgiou,"Jdeodorant:Identification and Removal of Type-Checking Bad Smells,"Proc.12thEuropean Conf.Software Maintanance and Reeng., pp.329-331,Apr.2008

[5] N.Tsantalis,A.Chatzigeorgiou,"Identification of Move Method Refactoring Opportunities,"IEEE Trans.Software Eng., vol.35, no.3, pp.347-367, May/June 2009.

[6] E.Van Emden and L.Moonen,"Java Quality Assurance by Detecting Code Smells,"Proc.Ninth Working Conf.Reverse Eng., pp.97-106,2002.

[7] S.Bouktif, G.Antoniol, E.Merlo, and M.Neteler, "A Novel Approach to optimize clone Refactoring Activit," proc. Eighth Ann.Conf.Genetic and Evalutionary Computation, pp. 1885-1892,July 2006.

[8] R. 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.