Refactoring Framework for Instance Code Smell Detection

D. Raj Kumar, G.M. Chanakya

***Abstract*— Code and design smells are the indicators of potential problems in code. They may obstruct the development of a system by creating difficulties for developers to fulfill the changes. Detecting and resolving code smells, is time-consuming process. Many number of code smells have been identified and the sequences through which the detection and resolution of code smells are operated rarely because developers do not know how to rectify the sequences of code smells. Refactoring tools are used to facilitate software refactoring and helps the developers to restructure the code. Refactoring tools are passive and used for code smell detection. Few refactoring tools might result in poor software quality and delayed refactoring may lead to higher refactoring cost. A Refactoring Framework is proposed which instantly detects the code smells and changes in the source code are analyzed by running a monitor at the background. The proposed framework is evaluated on different non trivial open source applications and the evaluation results suggest that the refactoring framework would help to avoid code smells and average life span of resolved smells can be reduced**

***Index Terms*—** *Software Refactoring, Monitor, Code smell detection, Instant Refactoring*

INTRODUCTION

Software refactoring [1] is a process of altering a software system in such a way that it does not change the external behavior of the code, and amends its internal structure. Refactoring is considered a best practice in creating and maintaining software, and research suggests that programmers practice it regularly. The term refactoring was introduced by Opdyke [2] in the year 1990. The basic idea of software refactoring can be drew back to restructuring i.e. the transformation of one representation form to another, while preserving the system's external behavior. Refactoring is becoming very popular due to "lightweight" development methodologies such as extreme programming that advocate continuous refactoring. Refactoring is basically the object-oriented variant of restructuring. In the evolution of software restructuring and refactoring are used to improve the quality of the software in terms of extensibility, reusability, and maintainability. Restructuring is needed to convert legacy code into a more modular or structured form or even to migrate code to a different programming language. In order to facilitate the identification of refactoring, experts have identified a number of typical situations that may require software refactoring which can be called as Bad Smells. Researchers proposed many algorithms to detect bad smells automatically. But detection algorithms are slightly different from existing ones. Detection algorithms minimize search scope to reduce human effort and save search time. Some existing detection algorithms may not reduce the time complexity because reduction becomes asymmetric. Tool support is critical for software refactoring [3]. Researchers have proposed tools to facilitate software refactoring. Refactoring tools are common in modern development environment. Hence refactoring tools cannot be applied until refactoring opportunities are identified by developers or detection tools.

2 TOOL SUPPORT

Although it is possible to refactor manually, tool support is crucial for the successful detection of bad smells. Manual refactoring is time consuming and error prone. Existing refactoring tools are essentially similar but they cannot be invoked before code smells are identified either by manually or automatically. A wide range of tools are available that facilitate software refactoring. Most main stream integrated development environment (IDE) such as Eclipse, Visual

***D. Raj Kumar**, Information Technology, SVEC, Chittoor, India.*
***G. M. Chanakya**, Information Technology, SVEC,Tirupati, India,*

Studio, and Intellij IDEA support software refactoring. Some of the refactoring tools are Resharper, Refactoring Browser, JRefactory, JBuilder. Refactoring tools should be driven by the developer, were the developer selects a particular source code entity which are to be analyzed. As software refactoring is human driven but the time and frequency of refactoring depends on software developers. Developers are attempting to improve the usability of refactoring tools by providing easy way of exploring source code when refactoring tools are active and by providing easy ways of selecting desired refactoring. The characteristics which affect the usability of refactoring tools are: automation, reliability, language independence, configurability. The degree of automation depends on refactoring activities and the extent to which each refactoring activities are supported. Reliability of refactoring tools depends on behavior preservation. Because most refactoring tools checks the precondition before applying it. Language Independence can be achieved by translating the code written in some language to other intermediate language, where the code can be restructured. Language Independence depends on Meta modeling and intermediate language. Configurability mechanism depends on usage of refactoring tools by adding/removing existing refactorings and smell specifications and by defining composite refactorings. Refactoring principles make refactoring tools more suitable for floss refactoring. Not all refactorings are supported by refactoring tools since detection and resolving bad smells remain time consuming even with the tool support.

3 RELATED WORK

Many works exists as the identification of problems in software testing, databases and networks Toure Mens and Toue [4] proposed five types of criteria to overcome existing research in software evolution and restructuring and they also define smell detection algorithms with logic rules in SOUL, a logic programming language. In order to avoid modal-configuration dialogs and to maintain the flexibility to configure a refactoring, E. Murphy-Hill and A.P. Black [5] used a tool in Eclipse called Refactoring Cues which presents configuration options. To use Refactoring Cues, Eclipse used to display a palette of refactorings adjacent to the program code. Because the refactoring view is not modal, we can use other development tools at the same time. This tool can help you select the code, and satisfy the floss refactoring principles. Travassos et al. [6] defined a process based on manual inspection to identify smells. But the identification of bad smell is essentially manual. Naouel Moha [7] proposes a Domain Specific Language (DSL) to specify bad smells.The DSL is based on an in-depth domain analysis of smells. Specifications in DSL are performed using a DSL in the form of rule cards. Marinescu [8] defined a metric based approach to detect code smells with detection strategies which captures deviations from design principles. Similarly Van Rompaey et al. proposed a metric-based approach for the detection of two test smells: general fixture and eager test. Simon et al define a distance-based cohesion metric, which measures the cohesion between attributes and methods. These metric aims at identifying methods that are used by more features of another class than the class that they belong to, and attributes that are used by more methods of another class than the class that they belong to. In 2012 M. Kim, T. Zimmermann, N. Nagappan [10] proposed a three-pronged view of refactoring is used in a large software development organization through a survey, interviews, and version history data analysis. In order to determine a de-facto definition and the value perception about refactoring, a survey is conducted with over three hundred professional software engineers, to examine whether the perception matches reality and whether there are visible benefits of refactoring. Hummel et al. [11] propose an index-based clone detection algorithm. The algorithm is not only incremental, but is also distributed and scalable. These incremental clone detection algorithms make the instant detection of clones possible, which consequently makes instant refactoring feasible. Specially designed detection algorithms for smells other than clone are also available. In 2010 G.Bavota, A.D. Lucia, and R. Olive [12] proposed a DECOR (DEtetion and CORrection) method to detect the code smells and to improve the quality of software system, and reduce the overall cost of development and maintenance. This DÉCOR method describes the specification and detection of code and design smells, DETEX allows specifying smells at high level of

abstraction and generating detection algorithms, and validating DETEX using precision. However refactoring tools are provided independently, and thus have to be invoked manually by software engineers. Consequently, these tools would not drive developers to refactor. Instead, such tools are driven by developers. Software engineers, especially inexperienced ones, might not invoke code smell detection tools on their own initiative [13], which might negatively affect the frequency of software refactoring. Consequently, smell detection tools usually have a number of parameters that users might manually adapt according to their own view. However, the adaptation is challenging, especially for inexperienced software engineers.

4 BAD SMELLS IN CODE

Code smell is the symptom which indicates something wrong. Bad smell indicates the code should be reexamined. Knowing where to refactor within a system is quite a challenge to identify areas of bad design. These areas of bad design are known as "Bad Smells". Deciding when to start refactoring and when to stop is just as important to refactoring as knowing how to operate the mechanism of a refactoring. However, refactoring itself will not bring the full benefits, if we do not understand when refactoring needs to be applied. To make it easier for a software developer to decide whether certain software needs refactoring or not, Fowler & Beck gave a list of bad code smells. Code smell is any symptom that indicating something wrong. It generally indicates that the code should be refactored or the over al design should be re-examined.

4.1 Descriptions of smells

Duplicated Code: The simplest duplicated code problem is when you have the same expression in two methods of the same class.

Symptoms: Redundant Code

Solution: Extract Class, Extract Method, Form template Method

Long Method: Method that is too long, so it is difficult to understand, change. The longer a procedure is the more difficult it is to understand.

Symptoms: Too long method that is difficult to understand and reuse

Solution: Decompose conditional, Extract Method, Replace Temp with query

Large Class: Class that has too many instance variables or methods, duplicated code.

Symptoms: Too many instance variables or methods

Solution: Extract Class, Extract Interface, Introduce Foreign Method

Long Method is a method that is too long, so it is difficult to understand, change. The longer a procedure is the more difficult it is to understand.

Symptoms: Too long method that is difficult to understand and reuse

Long Parameter List: Long parameter list is a parameter list that is too long and thus difficult to understand.

Symptoms: A method with too many parameters that is difficult to understand

Solution: Introduce Parameter Object, Replace Method with Method Object

Divergent Change: Divergent change occurs when one class is commonly changed in different ways for different reasons

Symptoms: A method changed continuously

Solutions: Extract Class

Shotgun Surgery: Shotgun surgery is similar to divergent change but is the opposite

They are hard to find, and it's easy to miss an important.

Solutions: Move Method and Move Field

Feature Envy: A classic smell is a method that seems more interested in a class other than the one it actually is in

Symptoms: Move method, Extract Method

Temporary Field: Temporary Field smell means that class has a variable which is only used in some situations.

Symptoms: A class has a variable that is only used in some situations

Solutions: Extract Class and Introduce Null Object

Refused Bequest: Refused Bequest smell means that a child class does not fully supported all the methods it inherits

Symptoms: A class could not support its inherited methods

Solutions: Replace Inheritance with Delegation

Data Clumps: Data Clumps smell means that software has data items that often appear together.

Symptoms: Data is always coherent with each other

Solutions: Extract Class and Introduce Parameter Object

Message Chains smell: Message Chains smell occur when we see a client that ask one object for another, which the client then asks for yet another object, which the client then asks for yet another object.

Symptoms: class that ask objects from one to another

Solutions: Hide Delegate

5 FRAMEWORKS

Refactoring framework consists of monitor, a set of smell detectors, refactoring tools, a smell view, and a feedback controller which instantly detects and presents code smells in a Smell View. An overview of the proposed framework is presented in Fig. 1

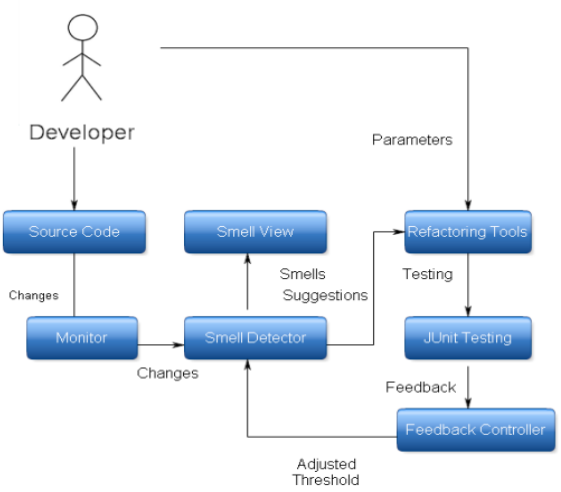

Fig. 1. Refactoring Framework

Refactoring frame work helps the programmer to analyze the code smells whenever source code changes occur. Detected smells in the code displayed through a smell view, which helps the developers to resolve detected code smells. The main task of the monitor is to analyze the changes that has been made on the source code, and decides which detection tools should be invoked .Whenever the monitor analyzes such changes it decides when and which refactoring tools should be used to avoid code smells. Generally Smell detection tools are resource consuming, therefore monitor minimize the frequency of invocation of smell detection tools under the assumption of timeliness. The smell detectors are called down by the monitor, once if the change has been identified the smell detector analyzes the changes and reports code smell through smell view. Smell view display bad smells in a friendly manner, such that the developers can avoid it and continue coding. The suggestions is displayed by the smell view will be used in modifying the changes that have been introduced into the code.

6 IMPLEMENTATION

To analyze the effect of refactoring against traditional refactoring, we had taken different types eclipse plug-ins such as PMD, Smell Checker, JDeodorant, and Check Style which detects the code bad smells. The evaluation was performed on different open source applications. InsRefactor was designed to detect the bad smells of java source code. It is an eclipse plug-in. For any java project in eclipse, Insrefactor would detect the code smells. It is also possible to detect code smells for single class or a method. Jdeodorant is an eclipse plug-in which identifies code smells and resolves them by applying corresponding refactoring technique. Jdeodorant detect only four types of code smells, namely God class, Long Method, Type Checking, feature envy. PMD is a source code analyzer which identifies potential problems like bugs, Dead code, and duplicate code. It is similar to find bugs. First, we had taken different types of eclipse plug-ins and given same java source code for each plug-ins to detect and resolve the bad smells once they appear. InsRefactor had detected eight types of c ode smells were as Jdeodorant had detected only four types of code smells consequently PMD and other plug-ins has detected six types of code smells. InsRefactor is a simulator which includes feedback controller which used to calculate the values of precision and recall were as other plug-ins does not contain such simulator. Once the refactoring was completed, the effort on different types of plug-ins and the quality of result were compared to check whether the recommended plug-in helped to simplify bad smell detection and resolution. The efforts include testing after refactoring to make sure that the applications works correctly. Smell View with refactoring suggestion and smells are represented in temporal order based on their appearance. Once the code free from bad smells the tool will generate an output as shown in figure 2

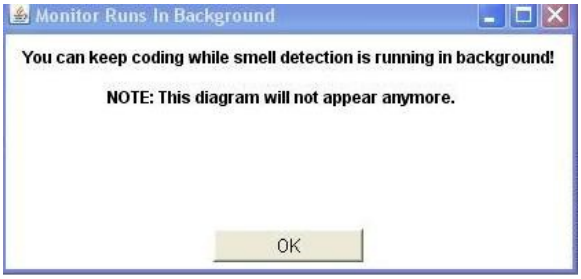

Fig. 2. Output of InsRefactor

Generally Bad smells in code are subjective, to perform optimization some detection algorithms generates thresholds value. Optimization can be done automatically by the framework.

Threshold = max (minThreshold, threshold);

Threshold =min (maxThreshold, threshold);

Delta= (minPrecision+maxPrecision)/2(curPrecision)

Threshold = threshold* (1+delta *10);

Precision is the number of smells detected by number of smells have been acknowledged and resolved.

$$Recall = (ES + DS) \div ES$$

$$Precision = (ES+DS) \div DS$$

Were ES=Existing Smell, DS=Detected Smell

7 RESULTS

Evaluation results are presented in table 1. Evaluation is performed on different non trivial open source application. Insrefactor plug-in able to detect 548 code smells out of 654 bad smells.

Type of Plugins	No of smells	No of smells detected	No of smells resolved
InsRefactor	654	548	500
JDeodorant	654	476	428
PMD	654	382	310
Others	654	448	390

Table 1 Evaluation of smells

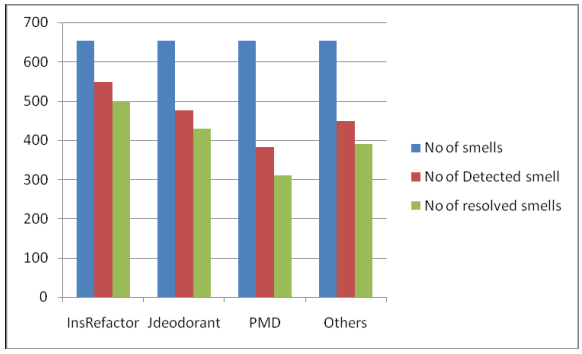

Fig. 3. smells detected by different plug-ins

InsRefactor helps to reduce the number of introduced smells by continuously warned by the refactoring framework. Refactoring framework has been implemented on eight types of code smells. We counted no smells being detected and resolved and also calculated the probability of smells being resolved. The following graph shows the probability of code smells that is being resolved.

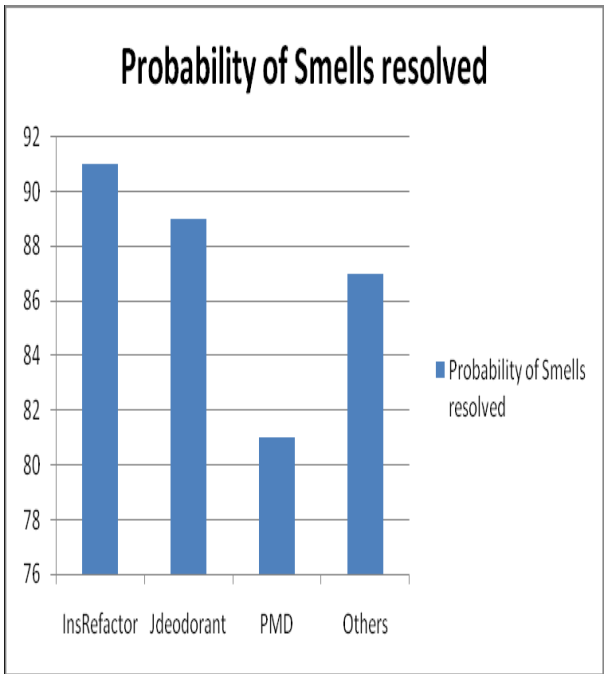

Fig. 4. Probability of smells resolved

The result of the evaluation depends on the amount of smells that has been detected, acknowledged and resolved.

8 CONCLUSIONS

Based on the study we propose a framework that can be evaluated based on number of applications. The main intension of the framework is to reduce the software cost and to improve software quality and to save the human effort. Proposed framework detects the eight kinds of bad smells. In order to improve performance and functionality of the framework we enhance more refactoring tools which used to identify more bad smells with various bad smells detectors. Future work is needed to evaluate a proposed framework based on more number of applications by both experienced and inexperienced developers. Removing more number of bad smells is believed to improve the quality of the software.

7 REFERENCES

[1] T. Mens and T. Touwe, "A Survey of Software Refactoring," IEEE Trans. Software Eng., vol. 30, no. 2, pp. 126-139, Feb. 2004.

[2] W.F. opdyke "Refactoring object oriented frameworks "PhD dissertation, Univ.of Illinois at Urbana-Champaign, 1992

[3] E. Mealy and P.Stropper, "Evaluating Software Refactoring Tool Support," Proc.Australian Eng., April 2006

[4] T. Mens and T. Touwe, "A Survey of Software Refactoring," IEEE Trans. Software Eng., vol. 30, no. 2, pp. 126-139, Feb. 2004

[5] E. Murphy-Hill and A.P. Black "Refactoring Tools: Fitness for Purpose" IEEE TRANSACTIOON SOFWARE ENGINEERING, VOL 30, NO 2, September 2008

[6] G. Travassos, F. Shull, M. Fredericks, and V.R. Basili, "Detecting Defects in Object-Oriented Designs: Using Reading Techniques to Increase Software Quality," Proc. 14th ACM SIGPLAN Conf

[7] N. Moha, Y.G. Guhneuc, and L. P., "Automatic Generation of Detection Algorithms for Design Defects," Proc. 21st IEEE/ACM Int'l Conf. Automated Software Eng., pp. 297-300, Sept. 2006

[8] M. Munro, "Product Metrics for Automatic Identification of 'Bad Smell' Design Problems in Java Source-Code," Proc. 11th IEEE Int'l Symp. Software Metrics, p. 15, Sept. 2005

[9] B. Van Rompaey, B. Du Bois, S. Demeyer, and M.ieger, "On the Detection of Test Smells: A Metrics-Based Approach for General Fixture and Eager Test," IEEE Trans. Software Eng., vol. 33, no. 12, pp. 800-817, Dec. 2007.

[10] M. Kim, T. Zimmermann, N. Nagappan "A Field Study of Refactoring Challenges and Benefits" ACM TRANSACTION ON SOFWARE ENGINEERING,VOL 30,NO 2,November 2012

[11] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt, "Index- Based Code Clone Detection: Incremental, Distributed, Scalable," Proc. IEEE Int'l Conf. Software Maintenance, pp. 1-9, Sept. 2010.

[12] G. Bavota, A.D. Lucia, and R. Olive" DECOR: A Method for the Specification & Detection of Code and Design Smells" IEEE TRANSACTION ON SOFWARE ENGINEERING, VOL 36, NO 1, September 2010

[13] E. Murphy-Hill, C. Parnin, and A.P. Black, "How We Refactor, and How We Know It," IEEE Trans. Software Eng., vol. 38, no. 1, pp. 5-18, Jan./Feb.2012

