INTERNATIONAL JOURNAL OF TECHNOLOGY AND COMPUTING (IJTC)
ISSN-2455-099X,
Volume 2, Issue 6 June 2016.

AN EMPIRICAL INVESTIGATION OF BAD SMELLS IN CODE ON MAINTENANCE EFFORT

Daljit Kaur [a], Rachna Rajput [b]
[a] *Research Scholar, M.Tech CSE*
[b] *Assistant Professor, CSE Department*
[a,b] *Department of CSE,* Guru Kashi University, Talwandi Sabo, *Punjab, India*

ABSTRACT

In this paper, we represent automated code smell detection and refactoring tools for calculating code quality by detecting Code Smells and increase code quality by Refactoring Techniques. Refactoring is a process for restructuring or improving internal structure of software without changing its behavior. To achieve this aim Declarative Programming approach is followed along with object-oriented software metrics. Detection of Code Smells is based on various Facts and Rules. We used this tool to detect the bad smells in oops based case studies. Code maintainability index is represented in three categories (Hi, Low, Medium) that shows source code quality (Low, Hi, Medium).

Keywords: - Code Smells, Code Quality, Maintainability Index, Detection of Code Smell, Refactoring, Object Oriented Metrics

I. INTRODUCTION

The term "code smell" [1] was firstly introduced by Kent Beck2 to define format problems in the source code of software which are examined by the experienced programmers. As written by Kent Beck: "A code smell is a clue that there is some fault in somewhere in your code [1]".

The built structure may not be causing any major impact (in terms of bugs and failures [2]) at the moment, but this fault must cause some serious problem on the fully build code source of the system and as a result of it, the quality factors shrinks. Code smells can clutter the design of a system, making it difficult to understand and maintain. Moreover, the presence of code smells gives birth to various new bugs [3] like huge development problems such as wrong architectural choices or even bad management practices.

In this Project our main target is on code smell and its automatic detection tools [7] which are developed. Code smells are mainly defined as a technique [5] which is used to check structural characteristics of software that point out a code or design (format) dilemma .It also make software difficult to learn and to maintain. This concept was introduced by Fowler [5], who derived 22 distinct types of smells. Later on, other authors (e.g. Mäntylä [MVL04]) found more smells, and new ones can are invented. Code smells are strictly related to the practice of refactoring software to capture its internal quality. As when the developers detect any kind of bad smells in code, they must solve it so that they can find out that whether the presence of smell is related to any fall in the structure of source code of system or not. If smell is found to be harmful for code then they choose accurate refactoring to remove it. As we all know ,problems are like a symptoms of any possible disease and to cure these diseases there are number of operations which are used to heal these problems so same conditions are applied on this concept of code smell and refactoring [9].

A software code [2] source becomes very difficult to maintain because it evolves with period of time. Also there designs are getting very complex and confused to learn so it is very important to reorganize the code time to time or for once. The very crucial thing is that when rearranging of code [9] is done it must be sure that the program behaves the same as older manner as it did previously before reorganization. Semantic preserving program transformations are known as refactoring. The process of refactoring [7] is not considered as a main front while development process because apply refactoring by hand having lots of drawbacks like error-chance and waste of time. It is very clear that the betterment of refactoring are not available for all the developers as it neither sum ups new features to the software nor improves any external software qualities. Although it is already defined that refactoring not having quality to hike the external software aspects but it has a important feature to help in betterment of internal attributes of source code of the software named reusability, maintainability and readability [18]. It is very challenging for the developers to find that which part of source code is profitable from refactoring even if they have very good grades in the study of refactoring. Many programmers learn from their experience, but for young generation developers it is very easy to understand and work on refactoring effectively as Martin Fowler and Kent Beck explained very well about refactoring in their books of refactoring [13].

II. LITERATURE SURVEY

All the researches on bad smell detection originate from the description of bad smells. Fowler defined the different 22 bad smells and also provided refactoring operations and methods to remove them [4]. Webster presented pitfalls in object-oriented development, and provided over eighty helpful summaries on how to avoid potential problems. Riel defined more than 60 guidelines to rate the integrity of a software design [1].

Brown described 40 anti-patterns for which he also provided heuristics to detect them. Moreover, there is also many more approaches in this field [2].

Author named karnam sreenu defined that software refactoring is a technique that transforms the distinct software artifacts to improve the software internal structure without affecting external nature [2]. Also it defined object oriented metrics can be calculated to detect the bad smell. In addition to metric-based approaches, there are some other peculiar approaches to detect bad smells.

Khomh proposed a novel approach to detect bad smells using Bayesian Belief Network [5]. Instead of just telling user whether or not a code component is affected by a bad smell, the approach calculates the impact probability.

Moha proposed Décor, which uses a Domain-Specific Language(DSL) [8] for specifying smells in high abstraction level, where software engineers can manually define the specification for bad smells detection using the taxonomy and vocabulary, and, if needed, the context of the analyzed systems.

Marija Katic, explains the main definitions and terms concerning software redesign is closely connected with the testing. This paper briefly presents the software redesign process and methods that are used in that process [10]. Although one can say that for example a source code redesign belongs to the implementation phase, tests are needed to ensure that the behavior is not changed.

Francesca Arcelli, Font ana Pietro Braione, Ma rco Zanonia [6] discovered that Code smells are structural characteristics of software that may indicate a code or design problem that makes software hard to evolve and maintain, and may trigger refactoring of code. Recent research is active in defining automatic detection tools to help humans in finding smells when code size becomes unmanageable for manual review.

Du Bois This paper analyzes how refactoring manipulate coupling and cohesion [7] characteristics, and how to identify refactoring opportunities that improve these characteristics. Coupling and cohesion on the other hand are quality attributes which are generally recognized as being among the most likely quantifiable indicators for software maintainability.

Pessoa this paper presents a tool that automates a technique for the detection and assessment of code smells in Java source code, developed as an Eclipse plug-in [8]. the detection technique used in the Smell checker tool is in contrast with all other known proposals due to the usage of a dynamic statistical process that relies on expert's knowledge that can be applied, theoretically to any smell.

In summary, most of the approaches available in the literature are solely based on structural information extracted from a single snapshot of software systems. As far as we know, there has not much research on the use of evolutionary data for code smells detection.

III. BAD SMELLS IN THE CODE

The concept of bad smell firstly explained by Fowler and Beck. The presence of any kind of bad smell in the source code simply means that there must be some serious issue in the code of the system. The bad smells are defined as any arrangement issue that makes source code of the system hard to capture and to maintain. Basically the bad smell in source code does mean that there is presence of any kind or bug or say error. Bad smell also does called any type of technical problem. Moreover, these all kinds of bad smells do not stop the code to execute but the only problem is that it make source code weak and useless. As result of presence of these the procedure of software's are getting slow also the chance of error is getting higher in future.

Code smell is like a clue card for developers that indicate them that there is something not good in the code. It generally motivates the programmers for the refactoring of code or reexamined the whole design ones again. The term appears to have been coined by Kent Beck .Usage of the term increased after it was featured in Refactoring. Bad code exhibits certain characteristics that can be rectified using Refactoring. These are called Bad Smells which are named as:

- **Long Parameter List**
- **Long Method**
- **Large Class**
- **Lazy Class**
- **Switch Statement**
- **Comment Lines**

Long Method: when method is too long means any method which have large number of coding lines.

Large Class: Classes that have large numbers of instance variables and large number of lines of code. Sometimes they are only used occasionally large classes can also suffer from code duplication.

Long Parameter List: Long parameter lists are hard to understand. Long parameter list means that a method takes too many parameters.

Comments: If the comments are present in the code more than the lines of code.

Switch Statements: Switch statements may produce duplication. You can find similar switch statements scattered in the program in several places.Maybe classes and polymorphism would be more appropriate.

Lazy Class: Classes that are not doing much work and number of method is null.

Temporary Field: when some of the instance variables in a class are only used occasionally.

Duplicate Code: The same code structure in two or more places is a good sign that the code need to be refactored.

IV. TOOLS FOR THE AUTOMATIC DETECTION OF CODE SMELLS

In this section we are going to explain all the previous detection tools which were already build to detect code smell from the code. We have Table 1 in which we are synthesizes some basic facts and features for the tools. Some tools, as we will see, have been developed to improve code quality during software development, other tools to support reengineering and maintenance activities. From all these defined tools , there are still some tools which are not capable to done refactoring of code automatically on detected smells (JDeodorant is the only one which provides refactoring choices), but modern IDEs are mainly able to perform refactoring automatically, also if they need user guidance. It would be desirable for smell detector tools to help the user to understand at least the cause of the smells and, as underlined in the guidelines proposed by Murphy-Hill and Black [MHB10], the tools should also not display smell information in a way that overloads the programmer, in case of smell proliferation.

Table 1 – Code smell detection tools

Tool	Version	Type	Analyzed languages	Refactoring	Link to code
Checkstyle	5.4.1 2011	Eclipse Plugin, Standalone	Java	No	Yes
DECOR	1.0 2009	Standalone	Java	No	No
iPlasma	6.1 2009	Standalone	C++, Java	No	No
inFusion	7.2.11 2010	Standalone	C, C++, Java	No	No
JDeodorant	4.0.4 2010	Eclipse Plugin	Java	Yes	Yes
PMD	4.2.5 2009	Eclipse Plugin, Standalone	Java	No	Yes
Stench Blossom	1.0.4 2009	Eclipse Plugin	Java	No	Yes

Checkstyle Checkstyle2 has been developed to help programmers to write Java code that adheres to a coding tandard. It is able to detect the Large Class, Long Method, Long Parameter List, and Duplicated Code code smells.

DECOR Moha et al. [MGDM10, MGM+10] defined an approach that allows thespecification and automatic detection of code and design smells (also called anti patterns). They specified six code smells by using a custom language, automatically generated their detection algorithms using templates, and validated the algorithms in terms of precision and recall. This approach is implemented in their Decor platform for software analysis.3 In the following, with the name Decor we mean the component developed for code smell detection.

inFusion inFusion4 is the current, commercial evolution of iPlasma. inFusion is able to detect more than 20 design flaws and code smells, like Duplicated Code, classes that break encapsulation, i.e. Data Class and God Class, methods and classes that are heavily coupled, or ill-designed class hierarchies.

iPlasma This tool [MMM+05] is an integrated platform for quality assessment of object-oriented systems that includes support for all the necessary phases of analysis, from model extraction, up to high-level metrics based analysis.5 iPlasma is able to detect what the authors define as code disharmonies, classified into identity disharmonies, collaboration disharmonies, and classification disharmonies. The detailed description of these disharmonies can be found in [LM06]. Several code smells are considered as disharmonies, e.g., Duplicated Code (named Significant Duplication), God Class, Feature Envy, and Refused Parent Bequest.

JDeodorant JDeodorant [TC11] is an Eclipse plugin that automatically identifies the Feature Envy, God Class, Long Method and Switch Statement (in its Type Checking variant) code smells in Java programs.6 The tool assists the user in determining an appropriate sequence of refactoring applications by determining the possible refactoring

transformations that solve the identified problems, ranking them according to their impact on the design, presenting them to the developer, and automatically applying the one selected by the developer.

PMD PMD7 scans Java source code and looks for potential problems or possible bugs like dead code, empty try/catch/finally/switch statements, unused local variables or parameters, and duplicated code. PMD is able to detect Large Class, Long Method, Long Parameter List, and Duplicated Code smells, and allows the user to set the thresholds values for the exploited metrics.

V. REFACTORING

Refactoring is like a new chapter on research and also not well defined as other topics. There are variety of explanations for refactoring but still now exact are on the way. Some definitions are referred below:

- Refactoring is the process of taking an object aim and re-arranging it in a variety of ways to make thedesign more flexible and re-usable. There are quite a few reasons you might want to do this, competenceand maintainability being possibly the most significant.
- To re-factor encoding code is to rewrite the code, to “clean it up”.
- Refactoring is the affecting of units of functionality from one place to another in your program.
- Refactoring has as a main objective, receiving each piece of functionality to exist in accurately oneplace in the software.

Refactoring Process

Refactoring process can be divided into a number of steps as shown below :

1. Classify the section of coding of software that where is the need of refactoring.
2. Decide which refactoring is required to perform to which section of the code of system.
3. Assurance that the applied refactoring conserves behavior.
4. Apply the refactoring [7].

5. Review the effect of the refactoring on the quality individuality of the software or the process.
6. Continue the consistency between the refectory program code and other software artifact.

IV. METRICS

As it is well know to everyone that for object-oriented program quality metrics is virtually endless (i.e. alone describes more than 200 complexity metrics), but m here going to target on only those program metrics which are frequently used like as number of methods, Cyclomatic Complexity, Number of Children, Coupling between Objects, Response for a Class and Lack of Cohesion among Methods.

Definitions for these metrics are:

- **Number of Methods:** it defines us how many methods are there in a class. It is also defined as a indicator of the functional size of a class.
- **Cyclomatic Complexity:** It is used to count the amount of all possible ways through an algorithm. It is an indicator of the logical complexity of a program which is depends on the number value of flow graph edges and nodes .
- **Number of Children:** It helps us to find the immediate descendants of a class. It is an indicator of the generality of the class.
- **Lines of Code:** It defines us how many lines of code is present in a method or in a class. It is an indicator of structural size of coding.
- **Comment lines:** It tells us the number of lines which are not used while execution of code.
- **Coupling between Objects:** It is a measure for the number of collaborations for a class. It is an indicator of the complexity of the conceptual functionality implemented in the class.
- **Response for a Class:** It is used to tell the number of both defined and inherited methods of a class, including methods of other classes called by these methods. It is an indicator of the vulnerability to change propagations of the class.
- **Lack of Cohesion among Methods:** It is used to tell the inverse cohesion measure (high value means low cohesion). Of the many variants of LCOM, we use LCOM1 as defined by Henderson-Sellers as the number of pairs of methods in a class having no common attribute references. It is an indicator of how well the methods of the class fit together.

VII. CONCLUSION AND FUTURE WORK

This paper represents refactoring filtering conditions, which help programmers (developer) to find applicant refactoring to reduce or remove Code Smells. These circumstances are analyzed at the program core level. They help programmers to know which part of program should be refactored. In addition, we arranged the rules

which help to select the refactoring technique that yields the highest maintainability. The experiment result implies that our advance technique can classify more effective refactoring than the refactoring books. Yet, our example covers only two refactoring (extracts method and restore temp with query). In our future work, we plan to manner an additional experiment that covers six refactoring for resolving long technique.

REFERENCES

[1] Beck, K. *Extreme Programming Explained: Embrace Change.* Addison-Wesley, Reading, MA, 2000.

[2] Karnam Sreenu 1, D. B. Jagannadha Rao2 "Performance - Detection of Bad Smells In Code for Refactoring Methods" International Journal of Modern Engineering Research (IJMER). Vol.2, Issue.5, Sep-Oct. 2012 pp-3727-3729

[3] Safwat M. Ibrahim1, Sameh A. Salem1, Manal A. Ismail1, and Mohamed Eladawy1 " Identification of Nominated Classes for Software Refactoring Using Object-Oriented Cohesion Metrics" IJCSI International Journal of Computer Science Issues, Vol. 9, Issue 2, No 2, March 2012 ISSN (Online): 1694-0814

[4] Veerapaneni Esther Jyothi1, Kaitepalli Srikanth2and K. Nageswara Rao3, "effective

[5] implementation of agile practices – object oriented metrics tool to improve software quality" International Journal of Software Engineering & Applications (IJSEA), Vol.3, No.4, July 2012

[6] Francesca Arcelli Fontanaa Pietro Braionea Marco Zanonia, "Automatic detection of bad smells in code: An experimental assessment ," Journal of Object Technology Published by AITO Association Internationale pour les Technologies Objets, c JOT 2011

[7] Stefan Slinger , "Code Smell Detection in Eclipse," Delft University of Technology

[8] Kwankamol Nongpong, "Integrating \Code Smells" Detection with Refactoring Tool Support " The University of Wisconsin-Milwaukee August 2012

[9] Panita Meananeatra and Songsakdi Rongviriyapanish," Using Software Metrics to Select Refactoring for Long Method Bad Smell" Computer and Information Technology Software Engineering Paper ID 118.

[10] Bart Du Bois and Serge Demeyer, Jan Verelst "Refactoring - Improving Coupling and Cohesion of Existing Code"

[11] M. Fowler, Refactoring: Improving the Design of Existing code. Addison-Wesley, 1999.

[12] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in 20th International Conference on Software Maintenance (ICSM 2004), 11-17 September 2004, Chicago, IL, USA. IEEE Computer Society, 2004, pp. 350-359.

[13] J. Riel, Object-Oriented Design Heuristics. Addison-Wesley, 1996.

[14] G. Travassos, F. Shull, M. Fredericks, and V. R. Basili, "Detecting defects in object-oriented designs: using reading techniques to ObjectOriented Programming, Systems, Languages, and Applications. ACM Press, 1999, pp. 47–56..

[15] F. Simon, F. Steinbr, and C. Lewerentz, "Metrics based refactoring," in Proceedings of 5th European Conference on Software Maintenance and Reengineering. Lisbon, Portugal: IEEE CS Press, 2001, pp. 30–38.

[16] E. van Emden and L. Moonen, "Java quality assurance by detecting code smells," in Proceedings of the 9th Working Conference on Reverse Engineering (WCRE'02). IEEE CS Press, Oct. 2002.

[17] M. J. Munro, "Product metrics for automatic identification of "bad smell" design problems in java source-code," in Proceedings of the 11th International Software Metrics Symposium. IEEE Computer Society Press, September 2005.

[18] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "A bayesian approach for the detection of code and design smells," in Proceedings of the 9th International Conference on Quality Software. Hong Kong, China: IEEE CS Press, 2009, pp. 305–314.

[19] M. Mantyla. Bad Smells in Software - a Taxonomy and an Empirical Study. PhD thesis, Helsinki University of Technology, 2003.

[20] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. L. Meur, "Decor: A method for the specification and detection of code and design smells," IEEE Transactions on Software Engineering, vol. 36, no. 1, pp. 20–36, 2010.

[21] T. Kamiya, S. Kusumoto, and K. Inoue, "CCfinder: a multilinguistic token-based code clone detection system for large scale source code," Transactions on Software Engineering, no. 4, 2002.

[22] Rao and K. Reddy, "Detecting Bad Smells in Object Oriented Design Using Design Change Propagation Probability Matrix", in International MultiConference of Engineers and Computer Scientists, 2008.

[23] R. Agrawal, T. Imielinski, and A. N. Swami, "Mining association rules between sets of items in large databases," in Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, 1993, pp. 207–216.