Study on Improving Efficiency of Software by Detecting and Correcting Code Smells

Suganya D, Kathiresan V, Gunasekaran S

ABSTRACT-Code smells denotes the poor standards of the implementation strategy. Presence of code smells makes source code maintenance a tedious process and also making proneness to faults and changes higher issue. Code smells are also something which results due to poor designing solutions called anti-patterns. These, code smell becomes a challenge for the software engineers to make out changes which might be a hindrance for the software and evolution of software. Hence, this survey paper focuses on various methods and techniques for improving the efficiency and functioning of the software. Code smells are defects in coding or design of a software which does not stop the software from functioning but it slows down the efficiency of the software gradually. It has a serious impact on the maintenance of the software in a drastic manner where the structural characteristics of software indicates a code or design problem that makes software hard to evolve and maintain which triggers refactoring of code. Code smells are suboptimal design choice degrading different aspects of the code quality indicating deeper design problems which causes problems in the evolution of a software product. Not all of them are equally problematic and some may not be problematic at all in some contexts.

Index Terms- Anti-patterns, refactoring, code smells, CRO, DETEX, DÉCOR, software evolution.

1. INTRODUCTION

In computer programming, Code smells are generally not flaws or errors or bugs, rather they are not incorrect technically. Currently, it does not stops the program from functioning but they indicate weakness in structuring or designing an application, This in turn makes the development slower, riskiness of bugs are increased which leads to failure in the future. Code smells occurs in both production code and test code. They are concepts by Fowler and Kent Beck suggesting that they are structural characteristics of software which makes software hard to evolve and maintain. Presence of code smells denotes bad designing and leads to less maintainable code and maintenance cost is increased. They are inflated due to poor software structuring, incomprehensible code, sloppy and error prone practices and inflexible design structures. The presence of smells in the code may degrade quality attributes leading to a higher likelihood of introduction of defects. In short, code smells or design smells are symptoms of potentially problematic code from software maintenance perspective.

- *D.Suganya is currently pursuing masters degree program in Software Engineering in Coimbatore Institute of Engineering and Technology, India,. E-mail: sgsuganya12@gmail.com*
- *Mr.V.Kathiresan is currently working in Computer Science engineering in Coimbatore Institute of Engineering and Technology, India,. E-mail:xyz.kathir@gmail.com*
- *Dr.S.Gunasekaran is currently working as head and Professor in Computer Science engineering dept in Coimbatore Institute of Engineering and Technology,India, E-mail gunaphd@yahoo.com*

However, they are only indicators of problematic code. The term code smells has come up with the way of helping the developers to recognize the codes which are defected and when those codes are need to be refactored.

2. METHODS FOR DETECTION AND CORRECTION OF CODE SMELLS

Marouane Kessentini et.al[2] suggest an approach which is automated for the need for generating rules to detect and correct defects. Code smells are generally not flaws, rather something which reduces the efficiency. This paper involves Genetic programming for automatic generation of rules to detect defects, whereas Genetic algorithm for correcting solutions. Blob, spagehetti code and functional decomposition are the code smells used for demonstrating this automated approach. It is arrived with the aim of detecting defects and correcting them as an optimization problem. Information used here are those from the previous projects called Defect examples. In detection step, design defects are detected by generating rules on the basis of software quality metrics. Set of base of examples, set of defect examples, set of quality metrics are inputs and set of rules are given as output which being generated. In correction step, the generated rules in detection steps are inputs including set of refactoring as output. For, this automated approach the symptoms are trained with fitness functions calculating each solution by comparison of detected defects. Since bad examples of code smells are

TABLE 1
TYPES OF CODE SMELLS

CODE-SMELLS	DESCRIPTION
Duplicated code	Identical code exists in more than one location.
Long method	Method or procedure that has grown too large.
Large class	Class that has grown too large.
Too many parameters	Long list of parameters is hard to read making calling and testing the function complicated.
Feature envy	Class using methods of another class excessively
Inappropriate intimacy	Class that has dependencies on implementation details of another class.
Refused bequest	Class that inherits a methods that it does not use at all.
Lazy class	Class that does too little.
Data Class	Classes with fields and getters and setters that do not implement any function in particular.
Data Clump	Clumps of data items always found together whether within classes or between classes.
Shotgun Surgery	Change in a class resulting in need to make a lot of little changes in several classes.
God Class	A class that takes too many responsibilities relative to the classes with which it is coupled.
God Method	A class having if at least one of its methods is very large compared to the other methods in the same class.
Blob	One large class monopolizes the behavior of a system while the other classes primarily encapsulate data.

used it will eventually lead to the usage of context-specific data, where it includes best/worst practices. Next, issue is rule generation which depends on detection results that are randomly generated.

Ali Ouni et al.[1] proposes an approach based on chemical reaction optimization developed for the purpose of detection and correction of code smells with respect to the prioritization schema including severity, importance an riskiness of code smells. They increase the rate of maintaining software and makes tedious in making the changes. Refactoring remains as an efficient way in removing code smells by changes made only internally without affecting the external behavior. When it comes to large scale systems fixing of code smells also remains at highest rate. So, prioritization of code smells can be done based on certain criteria which will be helpful in fixing the

code smells. This is done by an effective approach Chemical Reaction Optimization (CRO). Automated refactoring suggestions are for detecting and correcting the riskiest code smells that are to be prioritized. To make this efficient near-optimal sequence of refactoring from huge number of refactoring are done by CRO. Chemical refactoring optimization is used to find appropriate refactoring code smell while prioritizing most risky code fragments. Blob, data class, spaghetti code, functional decomposition, shotgun surgery, feature envy are the code smells involved for this approach. Set of inputs given are source code to be refactored, possible refactorings to be applied to list of code smells, bad smell detection rules, score of risk and severity for each of the detected code smells, preferences and prioritization of software maintainer. For these set of inputs, the set of output generated are optimal sequence of refactorings, specific refactorings list to improve software quality by reducing the number of detected code smells. The most important prioritization measures are formulating the refactoring tasks to correct code smells are severity, riskiness and priority. CRO designing is done with elements for code smells correction problem which includes solution coding in which specifically vector-based solution coding is used. Elementary reaction operators are used in further exploration in searching space for CRO. Simulations are done many times and their parameter settings has no general rules in determining them. Yet, correlation between correcting code smells are to be determined and evaluated by reducing the number of refactoring steps and improving its coherence.

Naouel Moha et al[3] suggest code and design smells are poor solutions in designing problems and those which also steps the evolution of a system. Methods are been used for detection of code smells contributed are DECOR, DETEX and Validation DETEX. By using DETEX, four code smells used are antipatterns Blob, functional decomposition, spaghetti code and Swiss Army Knife are for their automatic detection of algorithms. Code and design smells are not bugs, but something which hinders the evolution of the system from functioning. Fowler presented code smells whose structures in source code which suggests the possibilities of refactorings. Few smells in designing are duplicate code, long method, large class and long parameter lists. The cost of maintenance and development phase can be reduced in case of the code smells are detected. In large scale systems, the detection of code smells will be a very tedious work ad it becomes more time and resource consuming. Code smells are classified into two main categories inter and intra class and it is further subdivided into structural for static analysis, lexical for NLP and measurable for metrics. DÉCOR- Detection and CORrection method which constitutes all the necessary steps for specifying and detecting the code smells. Various steps are included in this algorithm. Description analysis is for text based description of smells. Specification constitutes vocabulary and processing translates the algorithms. Detection of code smells are done with operational specification and validation with source code with suspicious code constituents. DETEX-DETection EXpert helps software engineers by unified vocabulary with high level abstraction and along with domain specific language. Various DETEX steps are in terms of precision and recall. Domain analysis which is text based description of smells. Specifications which has vocabulary and taxonomy of smells as input. Generation of algorithm takes rule cards of smells as input and detection algorithm for smells are generated. Detection takes detection algorithm and model of system to be detected is taken as input and smell specifications of the suspicious class to be code smells.

Xiaodong Li et al[5] introduces a technique with an aim of providing solution for scaling up PSO algorithm in order to solve the large optimization problem in large scale systems. CCPSO2 coevolving cooperative particle swarm optimization is proposed technique based on random grouping which relies on Cauchy and Gaussian distributions. This approach is mainly for high dimensional problems saying from 100 to 200 variables. Evolutionary algorithms mostly serves as a best algorithm as a best algorithm for optimization problems. PSO is for managing large scale optimization problems and comparatively CCPSO is performing at a very outperformed manner. CCPSO is being developed to CCPSO2 based on various new techniques to improve its efficiency. Initially PSO algorithm does not deal with velocity, instead for generating next particle swarm positions Cauchy and Gaussian distributions are employed. In this, inertia weight PSO and constricted PSO is used. Population diversity is maintained highly by using ring topology for defining local neighborhood for each of the particle's position. *lbest* PSO is working on ring topology enhances the convergence speed at a slower rate and makes *lbest* outreach higher than

gbest model. Random grouping is better than going for selection of particular group sizes for each set of iterations. CCPSO2 remains to work efficiently on only small optimization problems scaled upto 2000 real valued variables.

3. Conclusion And Future Work

Detection of code-smells includes the future work for finding out the solution for correcting the detected code-smells. After, the code-smells are prioritized based on their metrics and inter-smell relationships the code-smells, then solutions to correct the detected and prioritized code smells. The optimization problem is enhanced using PSO in detection of code smells. Code smells are detected and prioritized based on the inter-relation between them and using certain metrics. The problem decreasing the quality of software and increasing the effort for maintenance of the software is being optimized efficiently by the detection of code-smells. Detection of code-smells are based on the metrics for each code-smells and they are prioritized using the prioritizing factors. Inter smell relation is also most significant feature in detection and prioritization of code smells ,since the effect of coupled interaction between the code-smells affects more than the individual code-smells. So, by detecting and prioritizing the code-smells the efficiency of the software can be maintained at a high rate. So, future work includes in detection of code smells by optimizing the problem. Further, code smells are detected based on the metrics of each of them and prioritization of the list of code smells are required based on different population sizes and for high dimensional functions. It is better to be used on multi-modal functions. CCPSO2 is enhanced with an ability to manage high dimensional

criteria like risk and importance of classes. Finally, the detected code smells are prioritized based on the inter-smell relations among the code smells.

References

[1] Ali Ouni ,Marouane Kessentini ,Slim Bechikh , Houari Sahraoui 29 April 2014. "Prioritizing code-smells correction tasks using chemical reaction optimization" Software Qual J

[2] Kessentini.M, W Kessentini, HA Sahraoui, M.Boukadoum, and A Ouni-2011 "Design defects detection and correction by example," in Proc. IEEE 19th Int. Conf. Program Comprehension, pp. 81–90.

[3] Moha,N, YG Gueheneuc, L Duchien, and AF.Le Meur-2010 ,"DECOR: A method for the specification and detection of code and design smells,"IEEE Trans. Softw. Eng., vol. 36, no. 1, pp. 20–36, Jan./Feb.

[4] Mika V Ma ntyla & Casper Lassenius-2006"Subjective evaluation of software evolvability using code smells: An empirical study" Empir Software Eng 11: 395–431.

[5] Li.X and X. Yao Apr- 2012 "Cooperatively coevolving particle swarms for large scale optimization," IEEE Trans.Evol. Comput., vol. 16, no. 2,pp. 210–224.

[6] Brown WJ, RC Malveau, WH Brown, and TJ Mowbray-1998 AntiPatterns:Refactoring Software, Architectures,and Projects in Crisis.Hoboken,NJ,USA: Wiley.

[7] Fowler.M, K.Beck,J.Brant,W. Opdyke, and D. Roberts 1999 Refactoring:Improving the Design of Existing Code. Reading, MA,USA:Addison Wesley