Refactoring System for Java Source Code

Vasudha Bhandari[1] Shimona Dubey[2] Akash Mishra[3] Ratodyasinh Parmar[4] Ms. Sarika Bobde[5]
[1,2,3,4]Student [5]Assistant Professor
[1,2,3,4,5]Department of Computer Engineering
[1,2,3,4,5]Maharashtra Institute of Technology Pune-411038, India

Abstract— Now a days due to the massive growth of online available open source codes and easily available help from the search engines tempts software developers to incorporate these codes into their softwares. Most of the times this is the reason for the code rotting, unhealthy dependencies between classes or packages, bad allocation of class responsibilities, over weight methods or classes, duplicated code, and many other varieties of confusion and clutter. Because of this, unwanted time and space complexities arises which eventually makes the software to work in below estimated level. This is actually the serious problem in the software industry and may play a vital role in software malfunctioning and other vulnerabilities. So as a tiny step towards this proposed model put forwards an idea of software code refactoring using graph dependency and Decision making techniques. The proposed model handles the complex coding structure related to Data members and member functions of the Java programming language efficiently.

Keywords: Dependency Graph, Feature Extraction, Decision making, Labelling

I. Introduction

Code Refactoring is one of the most widely used techniques for a seasoned developer. Refactoring is an essential concept that is used to restructure the entire code or parts of it without changing the underlying semantic understanding and code functionality. Refactoring is very useful and aids the developer by not changing the behavior of the code without introducing any bugs of their own as the changes are really small and inconsequential. Most of the times it is usually comprised of a set of basic, standard actions.

One of the most common reasons for refactoring the code is due to the fact that most of the code by developers are usually very dirty. Which mean it contains various error or a bad smell that can lead to the code being very difficult to update or maintain, which is something no developer wants their code to be. It affects the code and software life negatively. Most of the time the inconsistencies are not even recognized until a new feature addition break some functionality.

Majority of these inconsistencies are crept in due to crunched dates and unrealistic development cycles that are plainly unsustainable. Rushing a project will eventually lead to dirty code that cannot be handled properly. Fixing it is not an easy task, as the developer needs to specifically dedicate his/her time to refactor the code that would lead to a lot more delays in an already squeezed timeline.

Most of the time while writing code, developers contradict themselves and create a class that is dependent on itself, which is in turn dependent on the derived class. This leads to a lot of problems as it is generally not solvable as cyclic dependencies are very tricky and need a lot of workarounds to get it to work. Changing codes drastically can lead to a lot of problems into another set of codes dependent on these lines to malfunction.

When classes are dependent on each other, one class may be dependent on the results of another. This is problematic in scenarios where the output of the second one is crucial for the class to start operating normally. This is not possible as the former class is dependent on the output of the latter class. This is a vicious cycle that cannot be broken or made to work in any other way possible as these two classes are locked into a cyclic dependency.

Refactoring can provide some answers as it can cure the bad or dirty code. One of the functionalities of the automatic refactoring tools is to determine the dependency in the code and avoid the scenario of a closed or a cyclic dependency. As a cyclic dependency can be very crippling to get out of. Therefore, the refactoring avoids this fate by not allowing any cyclic dependencies to exist in the code.

Data Member Dependency is also a type of dependency that is very similar to the class dependency. This arises due to the data member being dependent on another data member. This is used to represent the flow of information throughout the code as the data members carry the information from one function to another. This is in a normal environment where the data members are dependent upon the input from the other function.

A cyclic dependency on a function might work if the member functions are exclusive and are centered around each other and performing a single operation. But it can be a nightmare to handle if the member functions are used somewhere else and as their dependency suggests, the code won’t be able to handle that scenario and lead to a catastrophic failure. To ameliorate these effects, code refactoring is highly essential.

Graphs are a class of data structures that are used to represent information in the form of nodes. The graph formation is not a linear process. The nodes are linked together to denote a certain relationship between the two data elements. The connections are made with lines called edges and can be used to connect any two nodes of the graph together.

Graphs in this implementation are used to depict the various variables, data members and member functions as the nodes and the various dependencies as the edges joining the various nodes of the graph together. this is used to easily process the various dependencies clearly and avoid and refactor them with utmost efficiency.

This research paper dedicates section 2 for analysis of past work as literature survey, section 3 deeply elaborates the proposed technique and whereas section 4 evaluates the performance of the system and finally section 5 concludes the paper with traces of future enhancement.

II. LITERATURE SURVEY

M. Badri [1] explores the concept of clone refactoring and its impact on the test code size for object-oriented languages. The authors have utilized linear regression in conjunction with various machine learning techniques, such as Random Forest, Naïve Bayes Theorem and the K-Nearest Neighbours, to develop an algorithm capable of predicting the impact. The data has been extensively collected with the help of an open source Java Based system, which has experienced clone-refactoring in one way or another. The technique has been quantified with the traditional techniques and achieves much higher efficiency.

J. Zhao [2] explains the multi-core era of computing that has put major multi-core CPUs in the hands of the common people at an unbelievably low price. It has ushered in the concept of parallel programming and more importantly, the OpenMP. Due to the rapid growth of data in this age of the internet has limited storage and computational abilities. Therefore, the authors have implemented a MapReduce module in conjunction with the OpenMP applications to compute parallelly in a cloud environment, thereby reducing the load on the computational and storage resources.

G. Kaur states that the management of code is one of the important skills that need to be possessed by the software developers and software architects. Therefore, to decrease the maintenance expenditure on the code; the code is refactored. Due to the increase in the size of the code with the number of features added, it needs to be counteracted in the form of code refactoring. The process of code refactoring increases the maintainability of the code and results in the reduction in the size which is actually beneficial for the software. The proposed algorithm performs as intended and have a lot more efficient than the traditional techniques. [3]

B. Lin [4] introduces the concept of naturalness of code, as a lot of research has been done in this area to conclude that like all other human languages, the source code is also predictable and highly repetitive. This confirms the relationship that a type of unnatural code is presumed to be buggy. The authors explain that this concept can be exploited in a way to achieve efficient code refactoring. As a buggy code has been deemed unnatural, the code refactoring can be easily done by maintaining the naturalness of the code. The technique has been unique and has outperformed state-of-the-art code refactoring tools.

M. Saca [5] expands on the concepts of Code Refactoring, by explaining the various steps that are required to achieve refactoring efficiently. The authors have diligently explained the various concepts and the general working of the code refactoring process in computers that have been designed poorly. Poorly designed systems have a lackluster performance, and code refactoring can greatly increase the performance and reduce the maintenance costs of the inefficiently designed computer system.

J. Zhao explains that due to the advancements in the area of Big Data has enabled a far more competitive era as the traditional storage systems are being discarded in the favor of the Big Data as it cannot keep up with the response speed and performance of the platform. Majority of the organizations are now moving their operations from their local machine to the cloud to benefit from the MapReduce improvements in their businesses. Therefore, the authors have developed a code refactoring mechanism for the sequential code to a MapReduce model for migration into the cloud. [6]

U. Devi [7] discusses the prevalence of code cloning as a major concern in the area of object-oriented languages. It has also been a problem for other methodologies, such as the SPL (Software Product Line), as its maintainability is one of the biggest concerns. SPL has been influenced a lot by the method of code cloning, the techniques for the analysis have inconsistent in their approach. Therefore, the authors have reviewed a lot of researches in this area to achieve maintainability with high forms of efficiency.

J. Vedurada [8] explores the concept of code refactoring as it is one of the most essential components for the maintainability of the software. Code is usually refactored to maintain the software and improve its readability all without hampering its original function. This is a very complicated and difficult thing to do, as one of the most important steps in the process is to identify the opportunities in the code for refactoring. The authors have done this with the help of "Replace Type Code with the state" and "Replace Type code with Subclass" in real time java applications as they are one of the most popular Java-based refactorings.

J. Kanwal investigates the prevalence of the practice of code cloning and how detrimental it is for the software. Usually, when the developers are refactoring various amounts of code while refactoring the code, they have a very different technique for handling the clones. To understand the workings behind software developer working on code refactoring, the authors have executed a longitudinal study. The results of the study indicate that clones of the code of the same class are consistently refactored and only a small set of code is refactored in a given time. [9]

G. Szoke [10] states that to maintain the quality of the code and prevent erosion, there is a need to keep refactoring the data. This is a very difficult and complicated process to refactor as the programmers have to identify potential areas in the code that need improvement and keep doing that constantly. But continuously refactoring the code has its own downfalls as the newer code has to be tested for inconsistencies and bugs. Therefore, the authors have proposed a technique for the purpose of refactoring with the help of code smells. The tool developed was tested rigorously and has been proven to perform efficiently.

C. Kulkarni [11] expresses that there has been widespread reuse of code in the area of software development. This practice is prevalent due to the fact that code reuse is one of the most effective ways to reduce the time taken. But there is a downside to this technique, as reusing code a lot of times, leads to a lot of code cloning, which leads to a negative impact on the maintenance of the systems. Therefore, the authors have developed a framework that determines if a clone code can be refactored to maintain the stability of the system with the proposed technique intact.

T. Sirqueira explains the occurrence of bad smells in code that occurs due to various inconsistencies in writing the code. Most of the techniques talked about in the paper have been defended by M. Fowler. Therefore, the authors have developed a tool to analyze and recognize the bad smells in a source code effectively. The tool has been used for evaluating academic codes and was aptly named the code analyzer. The tools were exceptionally beneficial for identifying and suggesting changes to reduce the impact of the bad smells from the code. [12]

Z. Chen [13] introduces developers that design mobile applications have to be well versed with new technology as soon as it arrives. Which is quite fast for a smartphone as there is a new one every year, the innovation never seems to end and the developers need to keep themselves up to date. This requires a lot of code reviews and this is very time-consuming for the developer as he can use this time to write more code. Therefore, the authors have conducted a survey for the change in code that reflected in the student's life. The authors concluded by stating that there is a need for a recommendation engine that can locate the errors and beg fixes efficiently.

III. PROPOSED METHODOLOGY

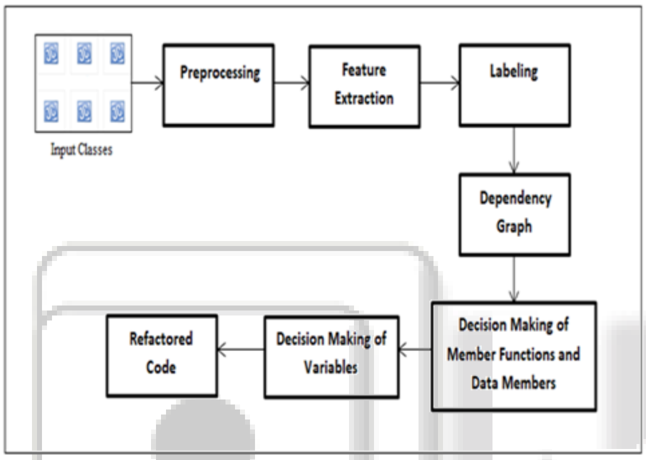

Fig. 1: Overview of the proposed methodology of source code refactoring technique

The proposed system of source code refactoring overview is depicted in the figure 1 and it is explained with the below mentioned steps.

A. Step 1: Preprocessing

This is the initial step of the proposed model, where the model is fed with a folder containing Java classes of a project. Once these classes are submitted, then these each and every classes are read in line by line manner and store in a list. After this each one of these classes from this list is accessed to perform preprocessing technique on them. By doing this preprocessing unwanted commented lines are shred off from the original code and braces are aligned properly in individual line, this eventually makes easy to process the classes in the coming future steps.

This preprocessing technique contains mainly two steps as mentioned below

1) Comment Eliminator

Here all the unwanted comments are eliminated from the source code, which are either starts with // or /* by tokenization process of Java.

2) Alignment of the Braces

This process aligns all the opening and closing braces of functions and control structures in individual line. Thereby line can contain only the braces that help to identify the boundaries of the control structures and functions.

B. Step 2: Feature Extraction and Labeling

This step eventually separates the features of a Java class in different labeled list. Majorly there are two features are extracting in this step.

1) Data Member identification

In this step data members of a class from the list of classes are segregated based on their position. This position is evaluated based on a general protocol of declaring the data members in the beginning of the class, which is being followed by the maximum number of Java developers around the world. This step actually identifies the presence of a member function in a class and thereby it decides the position of the data members to separate them in an individual list along with the class name.

2) Member Function Identification

Here each of the classes are traversing line by line to identify the member function based on the some protocols as discussed below. The member function line should contains '(' and ')' and next line should contain a '{'. As the line are traversing the number of opening brace '{' and closing brace '}' should be equal to form a member function and then all lines of member functions are collected in a list. This whole process can be depicted in the below mentioned algorithm 1.

1) Algorithm 1: Member Function Identification

```
// Input : $C_L$ [Class]
// Output : $MF_{SET}$ ( Member Function List)
Function : memberFunction($C_L$)
Step 0: Start
Step 1: $MF_{SET}$ =Ø, OBC=Ø, CBC=Ø
[ OBC: Opening brace Count, CBC: Closing brace Count]
Step 3: for i =0 to size of $C_L$
Step 4: IF $C_{Li}$ contains '( ' AND ' )'
Step 5: IF $C_{Li=i+1}$ Contains '{'
Step 6: START=i
Step 7: OBC++
Step 8: do
Step 9: IF $C_{Li}$ Contains '{'
Step 10: OBC++
Step 11: IF $C_{Li}$ Contains '}'
Step 12: CBC++
Step 13: while OBC!=CBC
Step 14: END=i
Step 15: for j=START to END
Step 16: $MF_{SETj=}$ $MF_{SETj+CLi}$
Step 17: end for
Step 18: end for
Step 19: return $MF_{SET}$
```

C. Step 3: Dependency Graph Formation

The Dependency of a data member and member function is estimated by evaluating the same in all other classes for their presence. This evaluation is being carried out by co-allocating the functions and data members of a class, along with their respective class or object to form the graph of dependency which later plays a crucial role in source code refactoring. This can be depicted in algorithm 2.

2) Algorithm 2: Dependency Graph Formation

```
// Input : $DM_L$ , $MF_L$,$C_L$
[ Data member List, Member function List, Class List]
// Output : $D_G$ Dependency Graph
Function : dependencyGraph($DM_L$ , $MF_L$,$C_L$)
```

```
Step 0: Start
Step 1: $N_{set} = ED_{set} = Ø$
[$N_{set}$ : Node Set , $ED_{set}$ : Edge Set ]
Step 2:$G(N_i,ED_i) = Ø$ ( Graph object)
Step 3: for i =0 to size of $DM_L$
Step 4: for j =0 to size of $C_L$
Step 5:  IF i!=j THEN
Step 6: IF $DM_{Li} \in C_{Lj}$
Step 7:  $G(N_i,EDi) = G(N_i,EDi) + G( CLi,CLj,EDi)$
Step 8: end if
Step 9: end for
Step 10: end for
Step 11: for i =0 to size of $MF_L$
Step 12: for j =0 to size of $C_L$
Step 13:  IF i!=j THEN
Step 14: IF  $MF_{Li} \in C_{Lj}$
Step 15:  $G(N_i,EDi) = G(N_i,EDi) + G( CLi,CLj,EDi)$
Step 16: end if
Step 17: end for
Step 18: end for
Step 19: return $G(N_i,EDi)$
```

D. Step 4: Source Code Refactoring through Decision making

Once the dependency graph is formed, the data members and member functions of classes are evaluated for their presence in the graph as the edges related to other classes. This is being carried out using the IF THEN rules of decision making. And this decision making is also used for the self-usage of the data members and member functions in the given class.

The dependencies of the variables belong to the respective member functions of a class are also estimated based on their respective weights. Once all the dependencies of the data members, member functions and variables are evaluated using the IF THEN rules, then source code is refactored and needed suggestions are shown to the developer.

IV. RESULTS AND DISCUSSIONS

The proposed technique of source code refactoring is deployed using windows based machine having a processor of Core i5 and primary memory of 6GB. To develop the model Java programming language is being used in Netbeans as IDE. The model is incorporated for the input of Java classes to handle the concept of source code refactoring. The ability of the developed software is being evaluated using some measuring indices like Root means Square Error (RMSE), which is elaborated as below.

RMSE is used to measure the error ratios between the expected outcomes and obtained outcomes. Lower values of the RMSE of the system indicate the higher accuracy of the model and vice versa. The RMSE can be measured with the below mentioned equation 1.

$$\mathrm{RMSE}_{fo} = \left[\sum_{i=1}^{N} (z_{f_i} - z_{o_i})^2 / N\right]^{1/2}$$

Where
$\sum$ - Summation
$(Zfi - Z_{oi})^2$ - Differences Squared
N - Number of samples or Trails

Trail No	LOC	Expected Member Function for Refactoring	Correctly Refactored Member Functions	MSE
1	54	2	2	0
2	89	3	3	0
3	205	5	5	0
4	389	6	5	1
5	550	12	11	1

Table 1: MSE for Member Functions

Trail No	LOC	Expected Data Member for Refactoring	Correctly Refactored Data Member	MSE
1	54	3	3	0
2	89	5	5	0
3	205	6	5	1
4	389	8	8	0
5	550	11	9	4

Table 2: MSE for Data members

Trail No	LOC	Expected Variables for Refactoring	Correctly Refactored Variables	MSE
1	54	1	1	0
2	89	3	2	1
3	205	5	4	1
4	389	11	9	4
5	550	14	11	9

Table 3: MSE for Variables

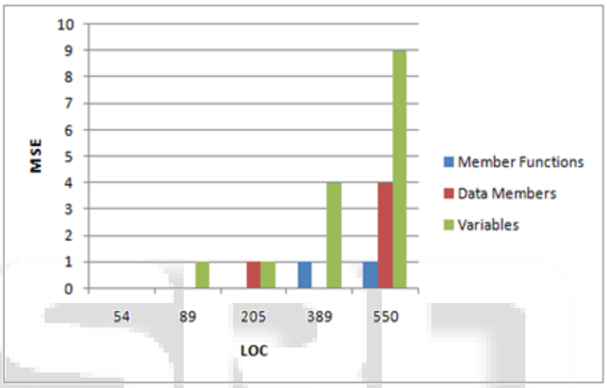

Fig. 2: MSE Comparisons

Member Functions	Data Members	Variables
0.6324	1	1.732

Table 4: RMSE for all three Entities

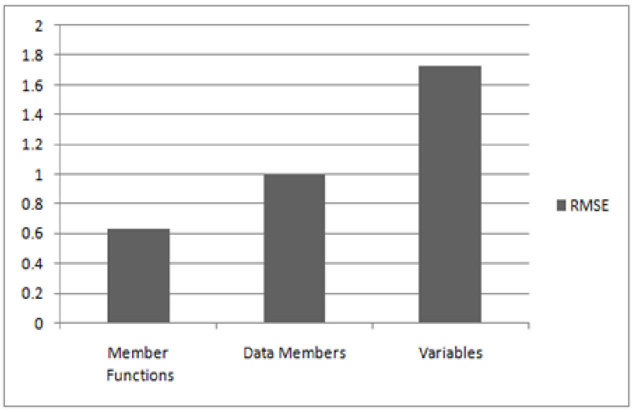

Fig. 3: RMSE Comparisons

RMSE for the refactoring of Member functions, Data members and Variables are estimated using Table no1,2, and 3. The Average Mean Square Error (MSE) for the member functions, data members and Variable are estimated as 0.4, 1 and 3 respectively as observed in the tables 1,2 and 3.

A plot is drawn in figure no 2 for the mean square error for these entities according to the 3 tables.

The plot in figure 3 represents RMSE of the three entities which is drawn based on the recorded values of the table 4. The figure 3 clearly indicates that proposed model yields an average RMSE of 1.21 that is really a good achievement in the very first attempt of this research article. This eventually shows the strength of the proposed model.

V. Conclusion and Future Scope

The Proposed model of refactoring the source codes of Java program mainly concentrate on three major issues in the generic coding. One - The uncalled member functions, two - unused data members and three- unused variables. These issues always add a considerable amount of complexities in the compilation of the code. So this research article mainly concentrate on handling three attributes based on the dependencies and obtain a very good RMSE of 1.21 for the conducted experiments that is a good sign of any beta version of research experiments.

In the future this research work can be enhanced to work many attributes like package dependencies, Inherited attributes, recursion functions, threads and many more to strengthen the concept of source code refactoring.

References

[1] M. Badri, L Badri et al, “Exploring the Impact of Clone Refactoring on TestCode Size in Object-Oriented Software”, 16th IEEE International Conference on Machine Learning and Applications, 2017.

[2] J. Zhao, M. Zhang, “Refactoring OpenMP CodeBased on MapReduce Model”, IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing Communications, BigData & Cloud Computing, Social Computing & Networking, Sustainable Computing Communications, 2018.

[3] G. Kaur and B. Singh, “Improving the Quality of Software by Refactoring”, International Conference on Intelligent Computing and Control Systems, ICICCS 2017.

[4] Bin Lin, Csaba Nagy, Gabriele Bavota, and Michele Lanza, “On the Impact of Refactoring Operationson Code Naturalness”, IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2019.

[5] Mauricio Saca, “Refactoring Improving the Design of Existing Code”, IEEE 37th Central America and Panama Convention (CONCAPAN XXXVII), 2017.

[6] J. Zhao, W. Wang, and H. Yang, “Code Refactoring Based onMapReduce in Cloud Migration”, IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, BigData & Cloud Computing, Social Computing & Networking, Sustainable ComputingCommunications, 2018.

[7] U. Devi, A. Sharma and N. Keswani, “A Review on Quality Models to Analyse the Impact of Refactored Code on Maintainability with reference to Software Product Line”, International Conference on Computing for Sustainable Global Development (INDIACom), 2016.

[8] J. Vedurada and V. Nandivada, “Refactoring Opportunities for Replacing Type Codewith State and Subclass”, IEEE/ACM 39th IEEE International Conference on Software Engineering Companion, 2017.

[9] J. Kanwal, K. Inoue and O. Maqbool, “Refactoring Patterns Study in Code Clones during Software Evolution”, IEEE 11th International Workshop on Software Clones (IWSC), 2017.

[10] GáborSz″oke, Csaba Nagy, Lajos Jen″oFülöp, Rudolf Ferenc, and Tibor Gyimóthy, “FaultBuster: An Automatic Code SmellRefactoring Toolset”, IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM), 2015.

[11] C. Kulkarni, “A Qualitative Approach for Refactoring of CodeClone Opportunities Using Graph and Tree methods”, A Qualitative Approach for Refactoring of CodeClone Opportunities Using Graph and Tree methods, 2016.

[12] T. F. M. Sirqueira, A. H. M. Brandl, E. J. P. Pedro, R. S. Silva, and M. A. P. Araújo, “Code Smell Analyzer: A Tool To Teaching SupportOf Refactoring Techniques Source Code”, IEEE Latin America Transactions, Vol. 14, No. 2, Feb. 2016.

[13] Z. Chen, “Helping Mobile Software Code Reviewers:A Study of Bug Repair and Refactoring Patterns”, IEEE/ACM International Conference on Mobile Software Engineering and Systems, 2016.