# Investigating the Effectiveness of Greedy Algorithm on Open Source Software Systems for Determining Refactoring Sequence

Sandhya Tarwani
SRM University, India
sandhya.tarwani@gmail.com

Ashish Sureka
Ashoka University, India
ashish.sureka@ashoka.edu.in

***Abstract*—The deeper problem in the source code are the bad smells that indicates something is wrong and if they are not detected timely, then they lead towards the complete deterioration of the working software causing major financial and productivity loss. Refactoring helps in removing these bad smells by improving internal quality attributes of the software without affecting its external behaviour. However refactoring needs to be applied in a controlled manner. In this study an approach has been propose for determining an optimal refactoring sequence that will maximize the source-code maintainability using greedy algorithm. The proposed approach selects the most optimum sequence at every step-in hope of finding the global optimum solution. We conduct an empirical analysis on four open-source software and select those classes that have bad smells greater than or equal to four. Further filtration is done by selecting those classes from the group that have high value of source code lines. We demonstrate the effectiveness of our approach using concrete examples of the experimental dataset and presenting summary results.**

***Index Terms*—Empirical Software Engineering, Software Quality, Refactoring, Greedy Algorithm, Software Maintainability**

## I. Research Motivation and Aim

Fowler [2] defined the term code smell. Code smells are indicators of root problem in the source code. They represent symptoms under which real problem exist and hence help software developers and maintainers to focus only on that portion of source code so that quality and reliability of the software gets improved. Code smells also indicate the violation of fundamental principle and negatively affect the quality of the software but they are not initially bugs. They eventually result in defects and failures only if they are not handled properly. Sometimes developers apply quick fixes to the problem identified and continue their work but it deteriorates the performance of the software and gets accumulated with time, hence chances of its termination increases. This not only wastes time but a lot of money and efforts are spent during the maintenance of the software. Hence, code smells need to be removed as soon as they are identified which in turn efforts and time saves and helps software to get complete and maintain within real time constraints. Refactoring is a term used for the restructuring and redesigning of the existing code without altering its external attributes. Fowler [2] defined more than 70 types of refactoring techniques like extract method, extract class etc., and all consists of several steps that will help maintainers to make original code less complex thereby increasing quality [4] [5] [8] [9] [12].

Although software consists of both functional as well as non-functional attributes but refactoring helps in improving its non-functional attribute like improving code readability, reducing complexity etc. It helps in transforming the source code that no longer contains code smell. For example, if long methods are present in source code than it results in poor understanding and increasing complexity. Therefore, it leads towards the need of applying refactoring techniques so that these problems get removed early. Techniques like Extract Method are used to extract some lines of code from the long method to make it short and precise so that it can be easily understood. Refactoring helps in improving the software maintainability. It needs to be applied in a short cycle and in a controlled manner so that it does not result in any negative impact on the quality and functionality of the software. Although these small cycles or transformation may not give extraordinary results but the cumulative effect will be very significant for the overall performance of the software [4] [5] [8] [9] [12].

Recently, several researchers are conducting study on finding the **correct or best sequence for refactoring techniques** so that software maintainability value gets enhanced [6] [7] [13]. If the sequence is known in advance to the software developers, then it will substantially reduce the effort and time spent on bug fixing and thereby improving quality of the software. In this paper, authors present their research work on finding the sequence of refactoring techniques by prioritizing the classes on the basis of number of bad smell and the value of line of code metric and thereby applying greedy algorithm. The refactoring sequence, thus, found will help in maximizing the software maintainability value measured using object-oriented software metrics [1]. While thee has been prior work on this topic (as discussed in the Related Work Section of the paper), the problem is relatively unexplored and not fully solved. In context to existing work, following are the *novel and unique contributions of the work presented in this paper*:

1) A greedy-approach based algorithm for determining the refactoring sequence for a software system. Our study is the first work on grouping classes based on the number

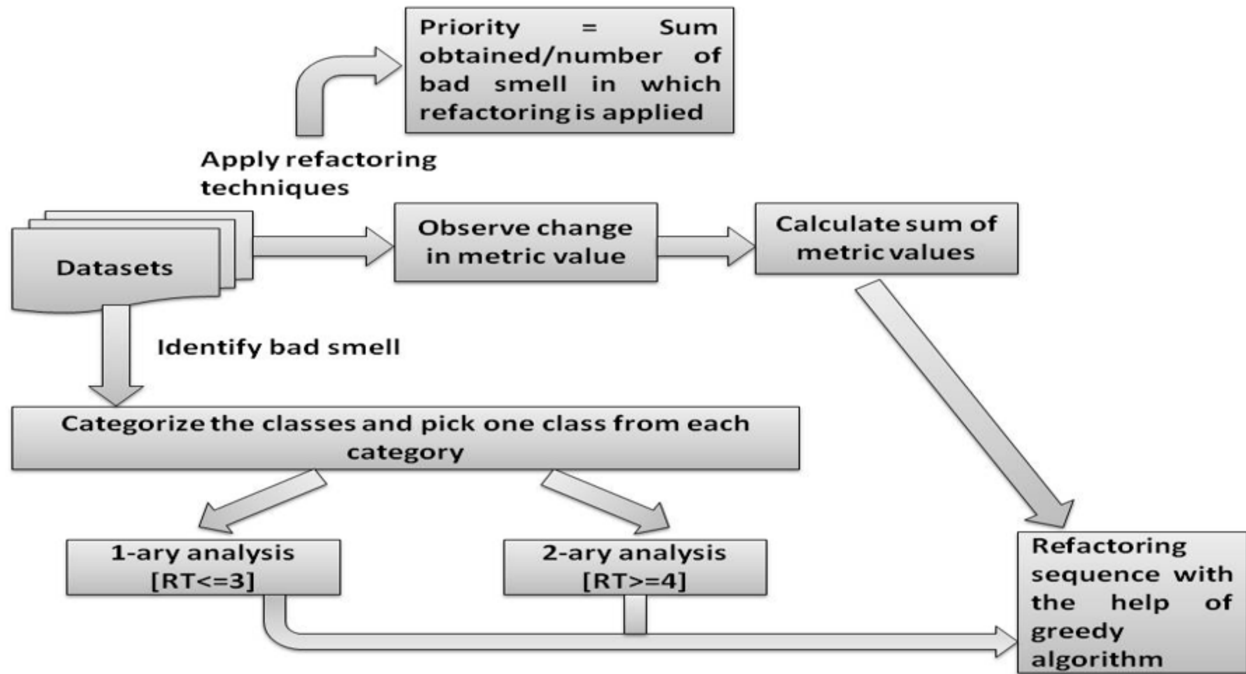



Fig. 1. Proposed Solution Approach - Multi-Step Process

of bad smells and then applying the greedy algoritm.

2) An empirical analysis on four open-source software systems to exhibit the effect of the proposed approach. Our study is the first work on JTDS, JChess, OrDrumbox and ArtOfIllusion dataset

## II. Related Work

A. Ghannem et al. [3] proposed an approach for automating the refactoring process in the source code with the help of Iterative Genetic Algorithm. It helps in integrating the feedbacks of users in classic GA and uses a fitness function which merges the analyzed design model and models from base of examples. Y. Khrishe and M. Alshayeb [5] conducted an empirical study to find out whether order of applying refactoring affects the quality of the software or not. They showed that applying refactoring in different orders have different impact on the quality of the software. I. Toyoshima et al. [11] proposed 3 gate refactoring algorithm which is a new refactoring algorithm that is developed with the help of three refactoring rules of Workflow net. A. Shahjahan et al. [8] used graph theory techniques to propose a new method of code refactoring which is applied on projects written in Java language. G. Szoke et al. [9] developed a refactoring toolset called FaultBuster that helps in detecting problems in source code with the help of source code analysis, running automatic algorithms to remove bad smells and execute integrated testing tools. Meananeatra et al. [6], Eduardo et al. [7] and Wongpiang et al. [13], Tarwani et al. [10] present techniques on searching for refactoring sequence for single class of a dataset.

## III. Proposed Solution Approach

We download source-code dataset from sourceforge and bad smells are identified with the help of plug-ins like JDeodorant [9] [12]. Figure 1 and 2 shows the flow-chart and the block diagram for the multi-step approach. Classes are then prioritized on the basis of number of bad smells and only those classes are considered whose number of bad smells are greater than or equal to 4. Accordingly, groups will be created based on the number of bad smells. Changes in metric values plays a key role and thereby taken into consideration after applying refactoring techniques so that sum of metrics can be calculated. In this study, trees are formed after applying refactoring techniques therefore, assigning priority to each technique is very important in the analysis for the formation of trees without any confusion. After assigning priorities, one class is picked from each category based on the higher LOC value. 1-ary tree is formed where number of refactoring techniques require is less than or equal to 3 and 2-ary tree is formed otherwise. After the formation of the trees, greedy algorithm is used to find out the best sequence for maximizing maintainability.

Greedy algorithm is an algorithmic paradigm that always makes choices that look best at that moment [13] [14]. It tries to make locally optimal choices at each stage in hope of finding the globally optimal solution. A global optimal solution is not guaranteed as it focuses and provides approximate global optimal solution for a given problem. Greedy algorithm works on the principle of choosing best at the moment and then solves sub-problems accordingly. It only considered the past solutions or present and will never focus on the future

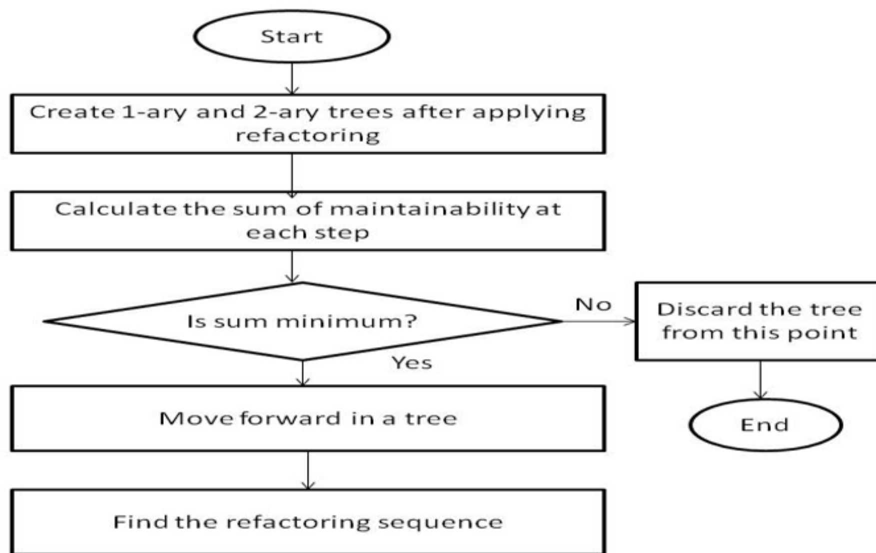



Fig. 2. Flow Chart and Sequence of Steps for the Proposed Approach

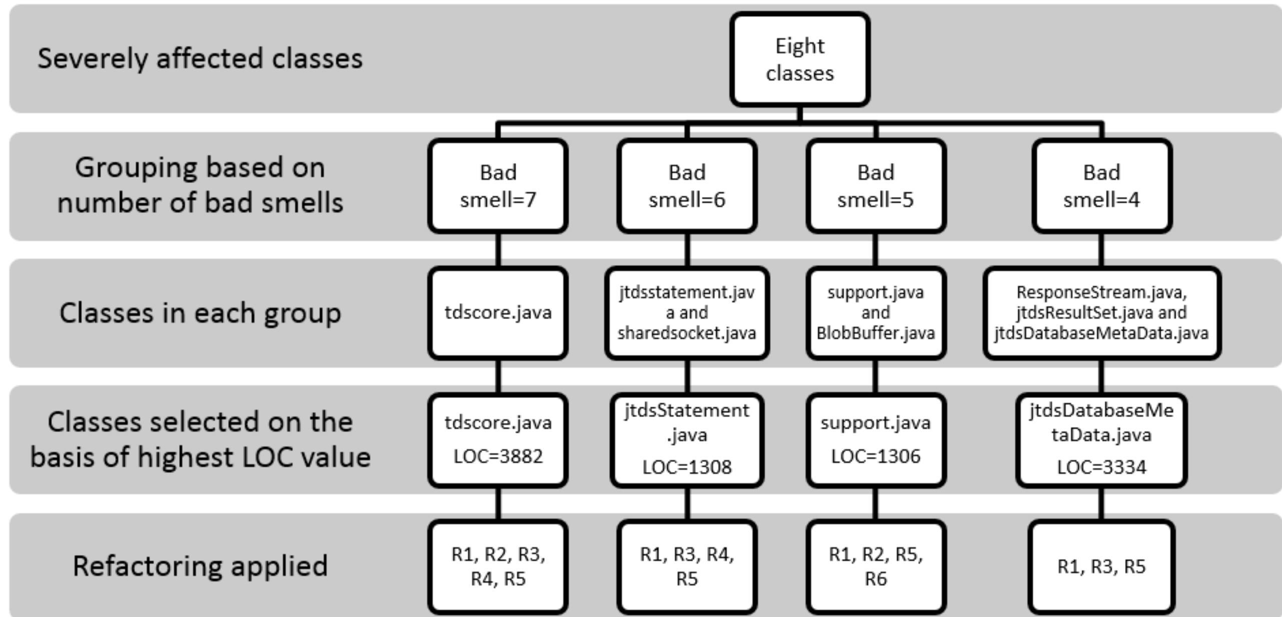



Fig. 3. Refactoring Sequence Determination Process for 8 Classes in JTDS project

solution of the sub-problem. It helps in reducing the bigger problem into smaller one by iteratively choosing best solution at each step [13] [14]. It is mainly used to solve the optimized problems that look for the best solution at a particular instant of time. It uses objective function that helps in the decision of choosing right solution such that given objective function gets optimized. The main point which needs to be kept in mind is that greedy algorithm will have only one chance to find the optimal solution as it never reverses its decision. Most of the programmers prefer greedy algorithm as it is most efficient one and analyzing its run time is very easy and time saving.

In this study, greedy algorithm is used to find the refactoring sequence for the datasets used. At every step, we move forward in the tree formed (refer to Figure 1 and 2), greedy algorithm takes decisions which are best at that moment and move forward and at the end it gives the refactoring sequence so that the overall maintainability can be maximized. Greedy algorithm plays an integral part in this study after the formation of the trees.

As it can be seen in Table I, various types of bad smells are identified in the source code and their removal requires different types of refactoring techniques. Table I displays bad smells found in eight classes of the Jtds project. Table I presents the number of bad smells, list of bad smells and

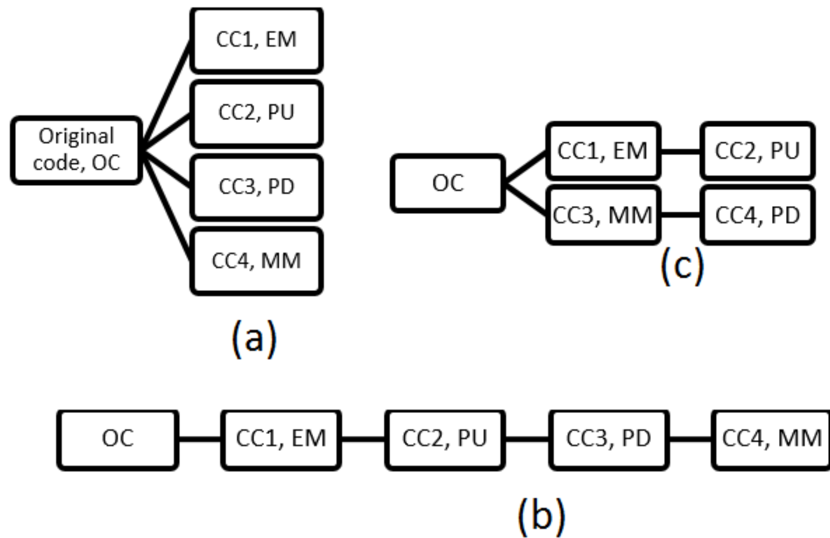
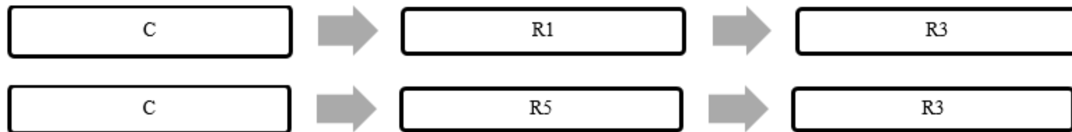



Fig. 4. Original Code (OC), Changed Code (CC) and Refactoring Techniques such as EM (Extract Mehtod) and trees such as 1-ary and 2-ary for a class in JTDS

**1-ary tree**

[jtdsdatabasemetadata.java]




**2-ary tree**

[Tdscore.java]

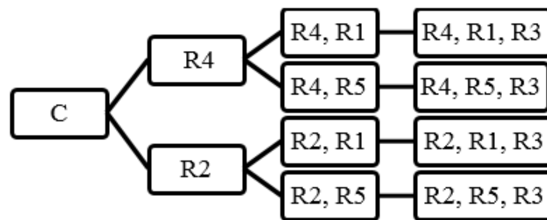



Fig. 5. An Illustration of 1-ary and 2-ary Tree Representation for Classes having Large Number of Bad Smells in JTDS Project

the lines of code in the Class containing the bad smell. We observe that there are classes with number of bad smells more than 5 which clearly shows the need of refactoring and also identifying a correct order of refactoring. Figure 3 shows the step-by-step procedure and result (as a worked out example and illustration) of applying the proposed approach on Classes belonging to the JTDS project. As shown in Figure 3, we perform grouping based on the number of bad smells, select classes based on the highest LOC value and then apply the refactoring sequence.

Different refactoring techniques can be applied to the same part of the code which will then convert the original code to different versions depending on which refactoring technique is applied. Sometimes, combination of refactoring techniques are applied to the code to see whether it result in the better version of the original code or not. These scenarios need to be kept in mind in forming the trees for different software. Trees represents structured object that needs to be explored by human beings from root node to the leaf. Trees are faster to search therefore are more convenient and efficient to use. In this study, trees are formed after applying refactoring to the original code in two ways that are discussed below.

TABLE I
EIGHT CLASSES FROM *jtds* PROJECT HAVING BAD SMELLS GREATER THAN FOUR

| | Classes | Bad Smell (BS) | # BS | LOC |
|---|---|---|---|---|
| 1 | TdsCore.java | God class, long method, type checking, feature envy, empty catch block, exception thrown in finally block, nested try statement | 7 | 3882 |
| 2 | JtdsStatement.java | Long method, type checking, feature envy, empty catch block, exception thrown in finally block, nested try statement | 6 | 1308 |
| 3 | SharedSocket.java | Feature envy, long method, god class, careless cleanup, empty catch block, exception throw in finally block | 6 | 971 |
| 4 | Support.java | Long method, god class, careless cleanup, empty catch block, nested try statement | 5 | 1306 |
| 5 | BlobBuffer.java | God class, long method, careless cleanup, empty catch block, exception thrown in finally block | 5 | 1207 |
| 6 | ResponseStream.java | God class, long method, feature envy, empty catch block | 4 | 463 |
| 7 | JtdsResultSet.java | God class, long method, feature envy, empty catch block | 4 | 1497 |
| 8 | JtdsDatabaseMetaData.java | Type checking, long method, empty catch block, exception thrown in finally block | 4 | 3334 |

TABLE II
REFACTORING TECHNIQUE, PRIORITY VALUE AND THE RANK ASSIGNMENT BASED ON EQUATION 1

| Refactoring Technique | Priority Value | Rank |
|---|---|---|
| R1 [EM] | 458.0313 | 5 |
| R2 [EC] | 429.755 | 3 |
| R3 [RC] | 529.1633 | 6 |
| R4 [MM] | 345.198 | 1 |
| R5 [RE] | 431.7525 | 4 |
| R6 [TEFB] | 385.06 | 2 |
| R7 [BOTB] | 0 | - |
| R8 [SC] | 0 | - |

TABLE III
OBJECT-ORIENTED METRICS VALUE AFTER APPLYING THE REFACTORING IN THE GIVEN SEQUENCE

| Sequence | CBO | LCOM | RFC | WMC | Ocavg | AHF | MHF | Total |
|---|---|---|---|---|---|---|---|---|
| **MM, EM** | 57 | 2 | 122 | 310 | 4.84 | 72.32 | 73.44 | 641.6 |
| **MM, RE** | 57 | 2 | 121 | 340 | 5.40 | 72.32 | 73.02 | 670.74 |
| **EC, EM** | 57 | 5 | 155 | 291 | 4.34 | 72.73 | 71.64 | 656.71 |
| **EC, RE** | 57 | 5 | 153 | 299 | 4.60 | 72.73 | 70.77 | 662.1 |

TABLE IV
OBJECT-ORIENTED METRICS VALUE AFTER APPLYING THE REFACTORING IN THE GIVEN SEQUENCE

| Sequence | CBO | LCOM | RFC | WMC | Ocavg | AHF | MHF | Total |
|---|---|---|---|---|---|---|---|---|
| **MM, EM, RC** | 57 | 2 | 117 | 297 | 4.64 | 72.32 | 73.44 | 623.4 |
| **MM, RE, RC** | 57 | 2 | 116 | 327 | 5.19 | 72.32 | 73.02 | 652.53 |
| **EC, EM, RC** | 57 | 5 | 150 | 278 | 4.15 | 72.73 | 71.64 | 638.52 |
| **EC, RE, RC** | 57 | 5 | 148 | 286 | 4.40 | 72.73 | 70.77 | 643.9 |

TABLE V
OBJECT-ORIENTED METRICS VALUE AFTER APPLYING THE REFACTORING IN THE GIVEN SEQUENCE

| Sequence | CBO | LCOM | RFC | WMC | Ocavg | AHF | MHF | Total |
|---|---|---|---|---|---|---|---|---|
| **MM, EM** | 33 | 9 | 103 | 133 | 2.56 | -167.74 | -34.62 | 78.2 |
| **MM, RC** | 33 | 8 | 100 | 124 | 2.48 | -167.74 | -40 | 59.74 |
| **RE, EM** | 33 | 8 | 109 | 151 | 2.80 | -167.74 | -37.04 | 99.02 |
| **RE, RC** | 33 | 8 | 106 | 142 | 2.73 | -167.74 | -42.31 | 81.68 |

### A. *One-ary analysis*

In one-ary analysis (refer to Figure 5), trees are formed by applying various refactoring techniques on a same potion of the code. It results in different version of the original code and hence will help in finding the best quality version among all. It needs to be kept in mind that the changed version should be an improved version of the original source code. After applying the refactoring techniques, this analysis will give the top most refactoring technique and hence will be very useful for the software developers to apply the best refactoring in the future during maintenance phase.

As shown in Figure 4, four refactoring techniques like Extract Method (EM), Push Up (PU), Push Down (PD) and Move Method (MM) have been applied to the initial original code to get four changed version of the software. The CC denotes the changed code and is collaborated along with the refactoring technique in a node of a tree. This notation becomes easy to understand and implement. Software developers will have four alternative paths and only one can b selected based on the improvement of the quality of the code with respect to maintainability. Hence, three refactoring techniques need to be eliminated and is done by considering the maintainability value of the software after applying it.

### B. *Two-ary analysis*

The two-ary trees (refer to Figure 5) are formed after combining refactoring techniques on the source code so that percentage of improvement of quality gets increased. It is important to enhance the quality of the software by applying refactoring in different combinations because refactoring is

TABLE VI
OBJECT-ORIENTED METRICS VALUE AFTER APPLYING THE REFACTORING IN THE GIVEN SEQUENCE

| Sequence | CBO | LCOM | RFC | WMC | Ocavg | AHF | MHF | Total |
|---|---|---|---|---|---|---|---|---|
| **EC, EM** | 41 | 7 | 29 | 179 | 12.79 | 100 | 100 | 468.79 |
| **EC, RE** | 41 | 7 | 27 | 183 | 15.25 | 100 | 100 | 473.25 |
| **TEFB, EM** | 55 | 11 | 46 | 266 | 9.76 | 100 | 100 | 587.76 |
| **TEFB, RE** | 55 | 11 | 44 | 270 | 10.77 | 100 | 100 | 590.77 |

TABLE VII
OBJECT-ORIENTED METRICS VALUE AFTER APPLYING THE REFACTORING IN THE GIVEN SEQUENCE

| Sequence | CBO | LCOM | RFC | WMC | Ocavg | AHF | MHF | Total |
|---|---|---|---|---|---|---|---|---|
| **EM, RC** | 10 | 128 | 220 | 267 | 1.44 | 100 | 100 | 826.44 |
| **RE, RC** | 10 | 128 | 220 | 275 | 1.40 | 100 | 100 | 834.4 |

itself expensive that can makes the situation even more worse, therefore controlled mechanism has to be kept in mind. As shown in Figure 3, Figure 4 and 5, the original code gets converted to the changed code CC1 by applying Extract method class. Afterwards push up refactoring technique is applied to get CC2 and so on. Different refactoring techniques are applied one after the other on the same portion of the code to get final changed version CC4. These refactoring techniques are applied in accordance to the presence of bad smell in code. Therefore at the end, refactoring sequence is found out to be EM → PU → PD → MM but consideration should be given to the fact that after applying this refactoring technique sequence in order, the value of software maintainability needs to be maximized then only there will be a gain in terms of time as well as budget.

It is not always possible to apply refactoring techniques one after the other. There may be situations where some will be eligible to apply and some are not. In this manner, two-ary trees are formed. As take another example where original code contains four types of bad smells and two types of refactoring techniques combinations can be applied on the same portion. Therefore, the two-ary tree for the same will be similar to Figure 5. As it can be seen, extract method along with push up can be used as well as move method and push down techniques can also be used. Here, there exist two refactoring sequences EM → PU and MM → PD. Hence, it depends on the value of software maintainability of the changed code version that will decide which combination improves the quality of the software more.

The order in which refactoring is applied initially depends on two factors, the presence of a particular bad smell that will help in judging the software developer which particular technique should be used in removing that smell and priority of the refactoring techniques, that is, Refactoring Priority, RP which is calculated with the help of maintainability values and number of classes in which a particular refactoring technique is applied. Initially, individual refactoring technique say $X$ is applied and value of maintainability, $M$ is observed. After dividing $M$ value with the number of classes in which $X$ is applied, priority of $X$ technique can be determined. In a similar way, priority values of all the refactoring technique

TABLE VIII
FOUR CLASSES FROM JTDS DATASET SELECTED AS ILLUSTRATIVE EXAMPLE TO DEMONSTRATE REFACTORING SEQUENCE DETERMINATION ALGORITHM

| Sequence | CBO |
|---|---|
| Tdscore.java (7 smells) | MM → EMRC |
| Jtdsstatement.java (6 smells) | MM → RC |
| Support.java (5 smells) | EC → RE |
| Jtdsdatabasemetadata.java (4 smells) | EM → RC |

used in this study is found out and highest priority is given to that refactoring technique that will have minimum value. It will help in selecting refactoring technique during formation of trees.

$$RP[X] = \frac{SO[M]}{NCR[N]} \quad (1)$$

In Equation 1, $SO$ represents the sum obtained and $NCR$ represents the number of classes in which refactoring is applied.

## IV. EXPERIMENTAL DATASET

Various developers and programmers across the globe are voluntarily creating open source software and uploading it on the platform where they are easily accessible to anybody for their use. With the help of this software, it becomes easy for researchers to generalize the results and work on these datasets so that a trend can be set up. Keeping this in mind, in this study four datasets have been downloaded from sourceforge.net. These datasets are chosen as their size varies from small to medium sized and hence will help in generalizing the result. The description of the dataset used has been provided below.

**JTDS**[1] - It is an open source jdbc 3.0 type 4 driver for Microsoft SQL Server. It is currently the fastest production ready JDBC driver for SQL server. It consists of 64 classes.

**JChess**[2] - It is a java based chess game project that requires two players playing on local computer or via network connection. It consists of 69 classes.

**OrDrumbox**[3] - It is an open source audio sequencer and software drum machine that is used to compose the bass line for completing the song. It consists of 217 classes.

**ArtOfIllusion**[4] - It consists of 739 classes and is a full featured 3D modeling, rendering and animation studio. It consists of subdivision surface based modeling tools, graphical language for designing etc.

For each metric of open source dataset which have been used in this study their descriptive statistics are collected like maximum value, minimum value, standard deviation etc but due to lack of space, tables are not included in this study. Therefore, following interpretations are added about descriptive statistics to give a little understanding about these datasets.

TABLE IX
JAVA CLASS, BAD SMELL PRESENT AND REFACTORING SEQUENCE

| | Java Classes | Bad Smell Present | Refacotring Sequence |
|---|---|---|---|
| | | **Jchess** | |
| 1 | game.java (4 bad smell) | God class, long method, type checking, feature envy | EM → MM |
| | | **OrDrumbox** | |
| 2 | RIFF32Reader.java (7 bad smell) | God class, long method, feature envy, empty catch block, dummy handler, careless cleanup, exception thrown in finally block | EC → MM → TEFB |
| 3 | song.java (6 bad smell) | God class, long method, type checking, feature envy, dummy handler, careless cleanup | EM → MM → TEFB |
| 4 | drumkitManager.java (5 bad smell) | God class, long method, dummy handler, nested try statement, careless cleanup | EC → RE |
| 5 | OrTrack.java (4 bad smell) | God class, long method, feature envy, dummy handler | EM → MM |
| | | **ArtOfIllusion** | |
| 6 | artOfIllusion.java (7 bad smell) | God class, long method, empty catch block, dummy handler, unprotected main, nested try catch, careless cleanup | EM → TEFB → BOTB |
| 7 | Scene.java (6 bad smell) | God class, long method, type checking, dummy handler, nested try catch, careless cleanup | RC → EC → TEFB |
| 8 | layoutWindow.java (5 bad smell) | God class, long method, type checking, feature envy, empty catch block | MM → EC → RE |
| 9 | TriMeshEditorWindow.java (4 bad smell) | God class, long method, type checking, feature envy | MM → EC |

TABLE X
NUMBER OF CLASSES AND CHANGES, NUMBER OF CLASSES FOR WHICH THE NUMBER OF REFACTORING IS GREATER THAN A DEFINED VALUE

| Feature | JTDS | JCHESS | ORDRUMBOX | ARTOFILLUSION |
|---|---|---|---|---|
| # Classes | 64 | 69 | 217 | 739 |
| # Changes | 37 | 57 | 86 | 194 |
| 1 RT | 8 | 8 | 49 | 248 |
| 2 RT | 6 | 2 | 44 | 144 |
| 3 RT | 6 | 0 | 25 | 75 |
| 4 RT | 3 | 2 | 7 | 39 |
| >5 RT | 4 | 0 | 6 | 39 |

TABLE XI
NUMBER OF CLASSES IN THE FOUR SOFTWARE SYSTEMS HAVING A PRE-DEFINED BAD SMELL

| Feature | JTDS | JCHESS | ORDRUMBOX | ARTOFILLUSION |
|---|---|---|---|---|
| God Class | 16 | 22 | 78 | 313 |
| Long Method | 26 | 22 | 78 | 390 |
| Type Checking | 5 | 3 | 12 | 111 |
| Feature Envy | 8 | 4 | 31 | 141 |
| Empty Catch-Block | 15 | 0 | 7 | 51 |
| Dummy Handler | 1 | 10 | 43 | 62 |
| Exception - Finally | 5 | 0 | 1 | 4 |
| Careless Cleanup | 5 | 1 | 8 | 22 |
| Unprotected Main | 1 | 3 | 3 | 3 |
| Nested Try | 6 | 1 | 5 | 15 |
| Over Logging | 0 | 0 | 0 | 0 |

The maximum value of AHF is nearly 100 in each datasets indicating hidden property of attributes. The standard deviation value of WMC is high in AOI and Jtds datasets indicating high faults. The standard deviation value of LCOM is highest in Jtds datasets hence cohesion among methods is minimum. The maximum value of RFC is highest for dataset AOI and Jtds which tells that these two datasets are more prone to faults. The mean value of CBO is highest in case of ArtOfIllusion dataset which implies that classes are highly complex therefore lacks understandability. The standard deviation value of Ocavg is highest in case of Jtds dataset which indicates complexity. The standard deviation value of MHF is highest in case of ArtOfIllusion dataset which shows that methods have little functionality as compared to other dataset's method.

## V. EXPERIMENTAL RESULTS

In order to demonstrate the workings of the proposed approach, one of the dataset results has been shown step by step for the clear and better understanding of the algorithm proposed to obtain refactoring sequence. Dataset *jtds* consist of 64 classes in which eight classes have been considered as severe as they contain bad smells greater than 4 and are shown in Table I along with their LOC values. Priority values of the refactoring techniques have been found out with the help of Equation 1 and accordingly rank is assigned as shown in Table II to each refactoring technique used to remove the bad smells present in the classes of the dataset. These ranks will help in selecting refactoring techniques for the formation of the trees. Classes are selected on the basis of highest LOC value as higher LOC value will lead towards more confusion and complexities and hence need to be focused urgently; therefore four classes have been selected as shown in Table VIII for finding out the correct refactoring sequence for maximizing maintainability. Table VIII also contains the type of refactoring techniques required in a particular class according to the different bad smells.

The formation of the tree is done according to the analysis discussed in the Section III-A and III-B. Refactoring techniques are selected on the basis rank assigned to them. Figure 5 describes the formation of the 1-ary tree for jtdsDatabaseMeta-

Data.java and 2-ary tree for tdscore.java respectively. Table III to Table VII contains the change in value of metrics after applying refactoring techniques so that sum can be calculated for all the metrics that will help in relating it with the maintainability of the class. Tables III to Table VII are created with the help of trees formed in analysis step. With the help of trees levels, each table is formed. If results for tdscore.java are taken into consideration than it can be seen from Table III that it is formed by after the application of R2 and R4 refactoring techniques. Afterwards, minimum value is selected among all the rows calculation by keeping in mind the inverse relation between the software metrics and maintainability.

In Table IV to VII, again the combinations of refactoring techniques are applied but this time only that part of the tree is taken forward that gets selected in table. For example, in table MM, EM combination has minimum sum of metric value therefore it is taken forward and only that part of the tree is selected and considered in future. Remaining part of the tree gets rejected and will not take part in obtaining the refactoring sequence as greedy algorithm is used that selects the optimal solution at each step. Therefore, at the end refactoring sequence for tdscore.java is found to be MM $\rightarrow$ EM $\rightarrow$ RC as this combination has minimum value of sum of metrics which results in maximum maintainability. In a similar way, refactoring sequence of other three classes is found out and is shown in Tables IV to VII and Table IX. The proposed algorithm is applied to other datasets as well and their respective sequences for severe classes are obtained as shown in Tables IV, VII and IX.

As it can be seen from the Table XI, the number of god class and long method type of bad smells are high in all the datasets which indicates the maximum use of extract class and extract method refactoring technique throughout the case study in removing them and hence it will be a part of refactoring sequence in most of the cases. Hence, it can be concluded that most common refactoring sequence will EC-EM. There are classes in the datasets in which bad smells are not present and hence no change is required. It can be seen from Table X, for instance consider dataset Artofillusion, 194 classes out of 739 classes will remain unchanged which indicates that no effort will be wasted in improving these classes and software maintenance team can focus on other classes with high number of bad smells so that refactoring sequence can be figure out by applying proposed algorithm. It can also be observed from Table XI that over logging type of bad smell is not present in any dataset and hence sprout class refactoring technique will never be a part of refactoring sequence.

## VI. Conclusion

We present an approach to determine the refactoring sequence for a software system having bad smells in several classes. The proposed approach is based on identifying bad smells for each class in the object-oriented software system (number of bad smells and types of smell) and grouping the classes based on the number of bad smells. The approach consists of computing object-oriented metrics and applying a greedy algorithms involving the formation of 1-ary and 2-ary trees searching for a sequence that improves the metric values and improves the maintainability of the system. We conduct experiments on four open-source Java projects to demonstrate the effectiveness of our approach. We conduct a series of experiments and explain the workings of the approach step-by-step using worked-out examples and finally present summary results for all the classes for all the four projects. We present descriptive statistics of the final results which shows that the proposed approach meets its desired objectives.

## References


[1] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

[2] M. Fowler and K. Beck. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[3] A. Ghannem, G. El Boussaidi, and M. Kessentini. Model refactoring using interactive genetic algorithm. In *International Symposium on Search Based Software Engineering*, pages 96–110. Springer, 2013.

[4] A.-R. Han and D.-H. Bae. An efficient method for assessing the impact of refactoring candidates on maintainability based on matrix computation. In *Software Engineering Conference (APSEC), 2014 21st Asia-Pacific*, volume 1, pages 430–437. IEEE, 2014.

[5] Y. Khrishe and M. Alshayeb. An empirical study on the effect of the order of applying software refactoring. In *Computer Science and Information Technology (CSIT), 2016 7th International Conference on*, pages 1–4. IEEE, 2016.

[6] P. Meananeatra. Identifying refactoring sequences for improving software maintainability. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 406–409. ACM, 2012.

[7] E. Piveta, J. Araújo, M. Pimenta, A. Moreira, P. Guerreiro, and R. T. Price. Searching for opportunities of refactoring sequences: reducing the search space. In *Computer Software and Applications, 2008. COMPSAC'08. 32nd Annual IEEE International*, pages 319–326. IEEE, 2008.

[8] A. Shahjahan, W. haider Butt, and A. Z. Ahmad. Impact of refactoring on code quality by using graph theory: An empirical evaluation. In *SAI Intelligent Systems Conference (IntelliSys), 2015*, pages 595–600. IEEE, 2015.

[9] G. Szőke, C. Nagy, L. J. Fülöp, R. Ferenc, and T. Gyimóthy. Faultbuster: An automatic code smell refactoring toolset. In *Source Code Analysis and Manipulation (SCAM), 2015 IEEE 15th International Working Conference on*, pages 253–258. IEEE, 2015.

[10] S. Tarwani and A. Chug. Sequencing of refactoring techniques by greedy algorithm for maximizing maintainability. In *Advances in Computing, Communications and Informatics (ICACCI), 2016 International Conference on*, pages 1397–1403. IEEE, 2016.

[11] I. Toyoshima, S. Yamaguchi, and J. Zhang. A refactoring algorithm of workflows based on petri nets. In *Advanced Applied Informatics (IIAI-AAI), 2015 IIAI 4th International Congress on*, pages 79–84. IEEE, 2015.

[12] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou. Jdeodorant: Identification and removal of type-checking bad smells. In *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*, pages 329–331. IEEE, 2008.

[13] R. Wongpiang and P. Muenchaisri. Selecting sequence of refactoring techniques usage for code changing using greedy algorithm. In *Electronics Information and Emergency Communication (ICEIEC), 2013 IEEE 4th International Conference on*, pages 160–164. IEEE, 2013.

[14] Z. Zhang, S. Schwartz, L. Wagner, and W. Miller. A greedy algorithm for aligning dna sequences. *Journal of Computational biology*, 7(1-2):203–214, 2000.