Improving Software Maintainability Using Optimum Refactoring Sequence

Sandhya Tarwani
University School of Information and Communication Technology
Guru Gobind Singh Indraprastha University
Dwarka sec-16, Delhi
+919650295744
sandhya.usict.007164@ipu.ac.in

Anuradha Chug
University School of Information and Communication Technology
Guru Gobind Singh Indraprastha University
Dwarka sec-16, Delhi
+919013766297
anuradha@ipu.ac.in

ABSTRACT
The surface indication of deeper problem in the source code are bad smells and if they are not removed in the early phases of the Software Development Lifecycle (SDLC) then they get accumulated making the source code complex and difficult to maintain. This is the reason why software maintenance is the most expensive phase. Refactoring is the process of removing these bad smells without altering the external attributes, but one should apply it in a controlled manner. In this study, we aim to propose a method in determining the refactoring technique sequence well in advance that will improve the maintainability value so that software maintenance team can complete their work within budget and time constraints. The basic approach used in every objective is to prioritize the classes with the help of proposed metric that are critically affected by the presence of maximum number of bad smells. The goal of this paper is to enhance the quality of the software and to reduce the maintenance cost by predicting the sequence with the help of i) Machine learning algorithms along with a new proposed metric that will help in determining the sequence ii) Various Meta-Heuristic algorithms and to figure out the best among all used iii) Deep learning algorithms impact on bad smells and refactoring sequences iv) Proposing a new hybrid approach to determine the sequence and v) Lastly, comparing all the methods and to figure out the best approach in terms of improvised maintainability value.

Categories and Subject Descriptors
D.2 [**Software Engineering**]: D.2.5Testing and Debugging, D.2.7Distribution, Maintenance and enhancement

Keywords
Software Refactoring; Maintainability; Quality; Bad smell; Maintenance; Heuristic; Deep learning.

1. INTRODUCTION
Evolution is the inherit property of the software system. With the introduction of new technology or changing requirement of the users, software is prone to changes which makes the source code more and more complex and hence difficult to maintain. In order to start the process of maintaining software, it is important for the team to find the design flaws and removes it as early as possible because they tend to accumulate with every phase of the SDLC and may jeopardize the working of the software. Program comprehension can be used to detect these design flaws as they understand the design of the software.

A design flaw, or as Fowler [6] introduce them as bad smells, are the surface indication of deeper problem in the source code and due to their presence, quality of the software may deteriorate continuously that includes its understandability or changeability. Fowler [6] introduces 22 bad smells along with refactoring technique to remove them. Refactoring is the process of removing these bad smells without altering the external attributes of the software. It is implemented in the source code by performing some changes in the code which implies a certain cost and risk. But, the presence of all types of smells are not harmful or problematic to the source code. Hence, it is important to note that refactoring should be applied in the controlled manner as the more changes are made in the software, there are more chances of increasing the defects in the software. Due to this reason, it is important to figure out the relationship between bad smells and maintenance problems.

It has been observed recently that existence of code smell can be used to evaluate the maintainability of the software. According to Institute of Electrical and Electronics Engineers (IEEE) standards [4], it is the ease with which a software system or component can be modified to correct faults, improve performance or other attributes. It involves continuous improvement by taking into consideration the past events and making decision with the help of experiences. Maximum amount of developer's effort is spent on the software maintainability phase of the lifecycle. Hence, efforts are being done by the researchers to reduce the cost and efforts of the maintenance team in maintaining the quality of the software.

As there will be numerous types of bad smell present in the source code, one type of refactoring technique will not be enough. Maintainers are required to apply various types of refactoring techniques one after the other in order to get rid of all the smells to clean the software. The maximum time is spent by the team in deciding which refactoring should be applied on the priority basis. Hence, the main aim of this paper is to find the most optimum refactoring technique sequence that will help in improving the maintainability value of the software. The sequence will be available to the maintenance team well in advance and whenever the software is under maintenance then, team can use that sequence to remove the bad smells present in the code while ensuring the quality of the software. This approach will reduce the efforts applied by the maintenance team and helps them in completing their work within budget and time constraints. For determining the refactoring technique sequence, authors have observed the need of systematic investigation. We hypothesized:

Through systematic investigation and prioritizing the classes of the software, we can detect the bad smell in the code and identify the refactoring sequences using i) Machine learning algorithms, ii) Meta-heuristic algorithm, iii) Deep learning algorithm and iv) A proposed hybrid algorithm. The best optimum sequence is figure out which provides the maximum maintainability value.

The evaluation of our hypothesis is done by answering the following research questions:

- RQ-1: What will be the best refactoring sequence using machine-learning algorithms? (Case 1)
- RQ-2: How meta-heuristic algorithms helps in determining the refactoring sequence? (Case 2)
- RQ-3: Which is the most promising Meta-heuristic algorithm that gives maximum maintainability value after the application of the refactoring sequence? (Case 3)

- RQ-4: How deep learning algorithms impact the detection of the bad smell and helps in determining the refactoring sequence? (Case 4)
- RQ-5: Can a hybrid algorithm be proposed in determining the sequence? (Case 5)
- RQ-6: Which type of algorithm is most effective in determining the refactoring technique sequence by maximizing its maintainability value? (Case 6)

We focused on the most critically affected classes of the source code that is being identified with the help of a metric proposed by Tarwani and Chug [14].

2. RELATED WORK

Our publication is closely related to the three aspects which includes detection of the bad smell, types of refactoring techniques along with its impact on the maintainability and determining the sequence of the techniques in order to enhance the maintainability of the software.

2.1 Bad smells

Bertan et. al. [10] focuses on the Aspect-Oriented Programming that aims to improve the software maintainability value and thereby presented an exploratory analysis of code smells in an aspect-oriented system. Their analysis suggests that newly-found code smells might occur more often than well-known ones. Bavota et. al. [2] fills the gap that are present in the bad test code smells study and presented this with the help of two empirical studies. These smells represent poor designed tests that affects the maintainability of test suites. The first study aimed at analysing the test smells in the source code and the other study is the controlled one that mainly focuses on the analysis of whether these smells affect the quality of the software during the maintenance task. They showed that these type of bad smells affects the comprehensibility of test suites negatively. Bryton et. al. [3] proposed a method for the detection of Long Method code smell objectively and automatically using Binary Logistic Regression model calibrated by expert's knowledge. Khomh et. al. [8] worked in determining the effect of presence of code smell on change proneness. They showed that classes which have code smells in them affects the change proneness more and hence the need to correct them as early as possible is required.

2.2 Refactoring and its impact on maintainability

Kataoka et. al. [7] proposed a quantitative evaluation method to check and measure the effect of refactoring on the maintainability enhancement. Tokuda and Batory [16] focused on three types of design evolution. They further evaluated whether refactoring technology can be transferred to the mainstream by restructuring non-trivial C++ application. Elish and Alshayeb [5] classifies the refactoring methods on the basis of its effect on software quality attributes which in turn predict the quality drift caused by refactoring. Bavota et al. [1] tried to fill the gap between the quality and refactoring by mining evolution history of three java projects and showed that there doesn't exists a clear relation between them.

2.3 Sequence of refactoring techniques

Wongpiang and Muenchaisri [17] used greedy algorithm to create sequences of refactoring techniques for optimizing the value of maintainability. Liu et al. [9] proposed a conflict-aware scheduling which schedules the refactoring techniques according to the conflict matrix and effects of individual refactoring using heuristic algorithm. Qayum and Heckel [13] proposed a method based on graph transformation to formulate refactoring locally. Ouni et. al. [12] proposed a multi-objective optimization approach to find the most optimum sequence that will maximizes quality and minimizes the semantic errors. Tarwani and Chug [15] proposed an approach to evaluate the sequence of refactoring using greedy algorithm which will enhance the quality of the software by maximizing maintainability. Morales et. al. [11] proposed an approach called RePOR (Refactoring approach based on Partial Order Reduction) and compare this approach with two well known Meta-Heuristic algorithms, a conflict-aware refactoring approach and a new approach based on sampling.

3. RESEARCH

3.1 Motivation

The main motivation behind this research is to find the most optimum refactoring sequence well in advance so that software maintenance team efforts gets reduced and cost of conducting the process also minimizes. The approach is to first prioritize the classes and to find the most critically affected class that contains the maximum number of bad smells. In the current study, authors have identified eleven types of bad smells and applied respective refactoring techniques as shown in table 1. Afterwards, various algorithms are applied to find the sequence that will maximize the maintainability value. Firstly, machine learning algorithms are applied and a new metric called TRI (Total Refactoring Index) is proposed. In the second phase, meta-heuristic algorithms are applied and comparison is done to check out which algorithm is better in terms of finding most optimum refactoring sequence that will simultaneously maximizes the maintainability value. Later on, a more advanced field will be chosen i.e., Deep learning algorithm and its effect on detection of bad smell will be observed. At the final stage, authors are planning to propose a hybrid algorithm that will efficiently find the optimum refactoring sequence automatically while yielding maximum maintainability value. The authors have selected mainly these three types of algorithm due to its certain advantages as shown in figure 2. The proposed methodology is being shown in figure 1.

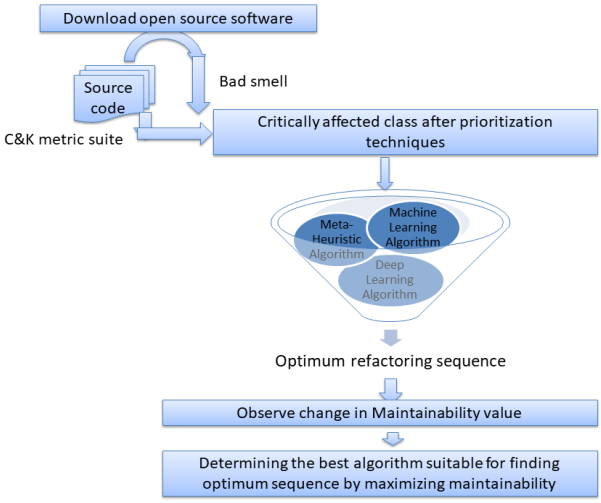

Figure 1. Proposed Research Methodology

Table 1 Bad smells identified along with its respective refactoring techniques

S. No.	Bad Smell	Refactoring technique
1	God class	Extract Class (EC)
2	Long Method	Extract Method (EM)
3	Feature Envy	Move Method (MM)
4	Type Checking	Replace Type Code with State/Strategy (RTS)
5	Empty Catch Block	Rethrow with exception (Re)
6	Dummy Handler	Rethrow with exception (Re)
7	Unprotected Main	Throw Exception in Finally Block (TEFB)

8	Nested Try Statement	Extract Method (EM)
9	Careless Cleanup	Rethrow with exception (Re)
10	Exception Thrown in Finally Block	Throw Exception in Finally Block (TEFB)
11	Over Logging	Sprout Class (SC)

3.2 Dataset Collection

Open source projects are mainly developed by various volunteers across the globe. Many studies have been conducted with the help of these systems as they help in generalizing the trend in Software Engineering field. Therefore, we have selected open source software to conduct this study.

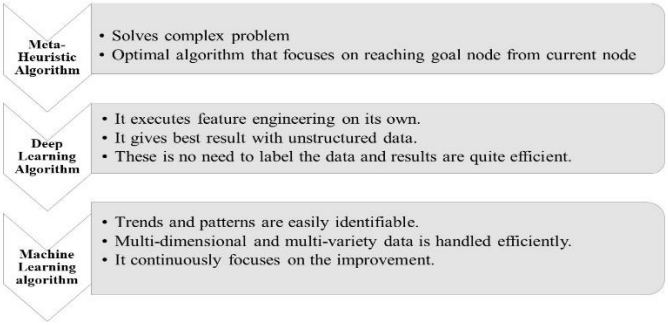

Figure 2. Advantages of Algorithm selected to find optimum refactoring sequence

3.3 Case 1: Sequence using Machine Learning Algorithm (Accepted in Journal of Information Processing System)

Methodology: We use six machine learning algorithm including Decision Tree Forest, Multi-layer Perceptron, Radial Basis Function Neural Network, Group Method of Data Handling, Support Vector machine and Linear Regression in order to find the refactoring sequence of four open-source object-oriented datasets. These algorithms were used to assign weight to the metrics used and a new metric called TRI (Total Refactoring Index) is proposed that helps in finding the refactoring sequence in advance which maximizes the maintainability value. Mean Absolute Error and Root Mean Square Error are used in this study and ten-fold cross validation is used in building the datasets for the above-mentioned algorithms.

Result: After applying Friedman test, it was observed that Decision Tree Forest is the most promising algorithm among others in determining the optimum refactoring sequence in every dataset used. For instance, one of the datasets (jTDS) maintainability value will improve only if the optimum sequence (Move Method→ Extract Class→ Replace Type Code with State/Strategy→ Big Outer Try Block→ Sprout Class→ Throw Exception in Finally Block→ Rethrow Exception→ Extract Method) is applied in this order. Similarly, sequence of all other datasets is also obtained. It has also been shown that in one of the datasets used (ArtOfIllusion) 94.9% of effort applied is saved with the help of this approach.

3.4 Case 2: Sequence using various Meta-Heuristic algorithm (Under review at Journal of System and Software and accepted in SUSCOM 2020 conference)

Methodology: We have used various Meta-Heuristic algorithms one by one to find the best refactoring sequence. The AO* algorithm is implemented first and is applied after the construction of multi-ary trees of the different datasets used. Afterwards, in another publication Hill-Climbing algorithm is used to figure out the optimum sequence. Meta-heuristic algorithms are used as they are efficient for solving maximizing and minimizing problem and helps in reducing the search space.

Result: It has been observed that maintainability value is improved in every dataset after the application of AO* algorithm in order to find most optimum refactoring sequence. For instance, 9.56% maintainability value is improved in jTDS dataset after applying ((Rethrow Exception AND Replace Type Code with State/Strategy) → (Move Method AND Extract Method) → Extract Class) sequence in order.

In another publication, authors have executed this problem with the help of Hill-climbing approach and it has been observed that 14.06% maintainability value is improved for jTDS software.

3.5 Case 3: Comparison of different types of Meta-Heuristic algorithm (In Progress)

Methodology: Heuristics are formal rules which uses searching techniques that are applied on the given problem to find acceptable and viable solution. We use four types of Meta-Heuristic algorithms i.e., Greedy, A*, AO* and Hill climbing on four open source dataset and compared all of them to find out the most promising algorithm that can be used which effectively maximizes the maintainability value and enhances the quality of the software.

3.6 Case 4: Sequence using Deep Learning algorithm (Proposed)

Methodology: We hypotheses the use of deep learning in determining the optimum refactoring sequence. Deep learning approach runs through several layers of neural network algorithms and each layer data gets simplified. This approach will help in automating our approach of identification of sequences and will further reduces the efforts. Deep learning consists of deep networks which uses unsupervised manner to learn features. It basically works without bringing engineering into the picture and with enough data and a good network architecture, neurons learn the abstract features. In this manner, we can just throw the problem domain in deep network constructed and sit back to get automated results. Hence, this approach is being taken into consideration in solving the problem of refactoring sequences.

3.7 Case 5: Innovation of Hybrid algorithm in order to find refactoring sequence (Proposed)

Methodology: We will propose a hybrid algorithm that will be implemented using MATLAB and will help in finding the sequence of open source dataset. We plan to use a pair of deep learning algorithm that will be most efficiently figure out the optimum refactoring solution in case 5. These algorithms will be well trained and a good architecture will be developed in order to find the most optimum refactoring sequence.

3.8 Case 6: Determining best approach to find sequence in order to maximize the maintainability value (Proposed)

Methodology: We will perform the comparison of all the algorithms on different open source dataset of varied size to figure out the most promising approach that can be used by the software maintenance team in order to complete their work with minimum efforts. Various parameters will be taken into consideration in order to select the most efficient algorithm.

4. TIMELINE

The author of the publication is the second year Ph. D student. She has completed Case 1 and Case 2 and submitted the work in Journal of Information Processing and Journal of Systems and Software respectively and they are currently under review. She also aims at submitting the remaining Case 3, 4, 5 and 6, in prestigious journals and conferences related to software engineering field. Figure 2, explains the timeline that will be followed by the authors in the course of its Ph. D course.

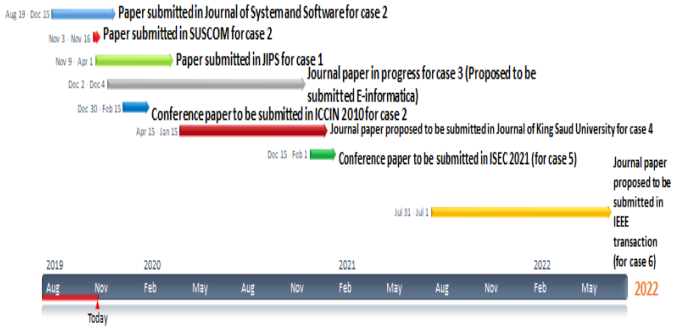

Figure 2. Proposed publication timeline

5. THREATS TO VALIDIY

While conducting this study, authors have faced various limitations as discussed below.

- Authors have used only open source datasets to demonstrate the idea of finding optimum sequence; but it can be applied to all the other software after further research.
- Further investigation is required to figure out the amount of refactoring techniques that needs to be applied as refactoring should be done in a controlled manner.
- The generalization of the result on other OO paradigm languages needs to be done as authors have used only java language-based results.

6. REFERENCES

[1] Bavota, G., De Lucia, A., Di Penta, M., Oliveto, R. and Palomba, F. 2015. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software*. 107, (2015), 1–14.

[2] Bavota, G., Qusef, A., Oliveto, R., De Lucia, A. and Binkley, D. 2012. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. *2012 28th IEEE International Conference on Software Maintenance (ICSM)* (2012), 56–65.

[3] Bryton, S., e Abreu, F.B. and Monteiro, M. 2010. Reducing subjectivity in code smells detection: Experimenting with the long method. *2010 Seventh International Conference on the Quality of Information and Communications Technology* (2010), 337–342.

[4] Committee, I.S.C. and others 1990. IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990). Los Alamitos. *CA: IEEE Computer Society*. 169, (1990).

[5] Elish, K.O. and Alshayeb, M. 2011. A classification of refactoring methods based on software quality attributes. *Arabian Journal for Science and Engineering*. 36, 7 (2011), 1253–1267.

[6] Fowler, M. 2018. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

[7] Kataoka, Y., Imai, T., Andou, H. and Fukaya, T. 2002. A quantitative evaluation of maintainability enhancement by refactoring. *International Conference on Software Maintenance, 2002. Proceedings.* (2002), 576–585.

[8] Khomh, F., Di Penta, M. and Gueheneuc, Y.-G. 2009. An exploratory study of the impact of code smells on software change-proneness. *2009 16th Working Conference on Reverse Engineering* (2009), 75–84.

[9] Liu, H., Li, G., Ma, Z.Y. and Shao, W.Z. 2008. Conflict-aware schedule of software refactorings. *IET software*. 2, 5 (2008), 446–460.

[10] Macia Bertran, I., Garcia, A. and von Staa, A. 2011. An exploratory study of code smells in evolving aspect-oriented systems. *Proceedings of the tenth international conference on Aspect-oriented software development* (2011), 203–214.

[11] Morales, R., Chicano, F., Khomh, F. and Antoniol, G. 2018. Efficient refactoring scheduling based on partial order reduction. *Journal of Systems and Software*. 145, (2018), 25–51.

[12] Ouni, A., Kessentini, M., Sahraoui, H. and Hamdi, M.S. 2012. Search-based refactoring: Towards semantics preservation. *2012 28th IEEE International Conference on Software Maintenance (ICSM)* (2012), 347–356.

[13] Qayum, F. and Heckel, R. 2009. Local search-based refactoring as graph transformation. *2009 1st International Symposium on Search Based Software Engineering* (2009), 43–46.

[14] Tarwani, S. and Chug, A. 2016. Prioritization of code restructuring for severely affected classes under release time constraints. *2016 1st India International Conference on Information Processing (IICIP)* (2016), 1–6.

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

[16] Tokuda, L. and Batory, D. 2001. Evolving object-oriented designs with refactorings. *Automated Software Engineering*. 8, 1 (2001), 89–120.

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