Software Metrics Supported Approach for Partial Reengineering Process

[*]Sumesh Sood, Anshul Kalia, Aayushi Kalia, Nikhlesh Kumar Badoga, Santosh Saklani

[*]Department of Computer Science, ICDEOL, Himachal Pradesh University, Shimla, Himachal Pradesh, India

Department of Computer Science, Himachal Pradesh University, Shimla, Himachal Pradesh, India

Department of Computer Science and Engineering, Thapar Institute of Engineering and Technology, Patiala, Punjab, India

Department of Computer Science, Himachal Pradesh University, Shimla, Himachal Pradesh, India

Department of Computer Science, Himachal Pradesh University, Shimla, Himachal Pradesh, India

Abstract

Reengineering is the process of assessment and modification of existing software to redesign it into a new form and the subsequent implementation of the new form. The emphasis lies on preserving the existing functionality while transforming into the new form. The metrics have a vital role in measuring the properties of software projects quantitatively through well-defined measurement rules. The reengineering process in this work has been supported by the software metrics-based approach. To make the reengineering process economical and efficient, the Rainfall Model has been used in different phases of the proposed framework. The measurements obtained by metrics are used to provide the basis for making decisions about reengineering requirements of software engineering tasks.

Keywords: Reengineering, Metrics, Maintainability Index, Cyclomatic Complexity.

1.0 Introduction

Reengineering is the process of assessment and modification of an existing software to redesign it into a new form and the subsequent implementation of the new form. The emphasis lies on preserving the existing functionality while transforming into the new form. The metrics have a vital role in measuring the properties of software projects quantitatively

through well-defined measurement rules. The measurements are used to provide the basis for making decisions about planning and performance of software engineering tasks.

The reengineering process in this work has been supported by the software metrics based approach. In order to make the reengineering process economical and efficient, the Rainfall model has been used in different phases of the proposed framework.

The research is relevant for all those professionals, academicians, and researchers who are working in the fields of software (maintenance) engineering, information economics and cognitive psychology. The subject of maintenance has high importance for information technology costs as well. Maintenance costs are often a high percentage of total information technology costs, ranging from 50 to 90 percent. Furthermore, customers do not buy new software every time if something needs to be changed. In most cases, changes made in the existing information system costs less. It is assumed that software with good overall structure and architecture should cost less for maintenance.

2.0 Literature Review

There are various researchers who have done a great amount of work in software reengineering and presented their work from time to time.

In 1985 Lehman and Belady emphasized on the organized notation of software evolution at IBM and since then it has developed into a recognizable field of research (Lehman and Belady, 1985). Later in the early 90's, the cynosure of the system development shifted to reengineering of legacy systems.

In 1994 Oman and Hagemeister introduced the Maintainability Index. The Maintainability Index is considered as one of the useful metrics, which is tested mathematically most of the time. It was tested in the field by Hewlett-Packard. From these field tests the many coefficients (the "magic numbers") in the equation have been found (Oman and Hagemeister, 1994).

Ahrens and Prywes in the mid 90's proposed a new model (LRSLC) for the life cycle of legacy software systems, which described the methods of reuse. The proposed model is considered as the generalized one, which is used to count the contribution of legacy software for the production of software components through reuse (Ahrens and Prywes, 1994, 1995).

MORALE, developed in Georgia Institute of Technology in 1997 for the purpose of reengineering. It handles the issues concerning design and evolution of complex software solutions (Abowd et. al, 1997).

Fahmy and Holt (2000) in their work categorize the architectural transformations that are useful during program maintenance (Fahmy and Holt, 2000). These include lifting and hiding transformations, diagnostic and sifting transformations, and repair transformations.

The L2CBD methodology presented in 2006 is able to provide a reengineering process including concrete procedures, product-work, guidelines and considerations. The L2CBD can transform legacy systems into new component systems with improved software architecture (Konda, 2005).

Singh and Sood in 2006 analyzed various reengineering methods and came out with four scenarios of software reengineering (Singh et. al, 2007). Out of these four scenarios, two scenarios require reengineering of some parts of the software and not the whole software. It leads to the need for partial reengineering of software systems.

Mishra et. al. 2009 has designed a model named as CORE in order to analyze and evolve reusable software components (Mishra, Kushwaha, and Misra, 2009). The reverse engineering techniques have been used to extract information of legacy systems concerning architecture and services. These services are converted into reusable components at a later stage of the system. The study lacks to describe the basis used for reengineering the components.

Anitha et. al in 2012 discovered the various risks involved in the transformation process of software reengineering. It suggested light weight process Extreme Programming as one of the methods to deal with the process transformation problem. The risks have been assessed quantitatively, which is helpful in reducing the transformation effort, increases the project cost, and improves the customer satisfaction (Anitha, Karthika, and Alagarsamy, 2012).

Duan et. al in 2013 conducted a study on a legacy software named ‘Automatic Test System’. It provided hierarchical software architecture as a method to address the issue of software reengineering. The study is also helpful in resolving resource invoking and semantic analysis issues through reengineering (Duan et. al, 2013).

Rather and Bhatnagar in 2016 proposed a software reengineering process model. The model focuses on the reuse and restructuring of project management activities. It highlights various advantages of reengineering (Rather, and Bhatnagar, 2016).

Majthouab et. al in 2018 proposed an integrated approach of redesigning the complete system from its SRS to testing. This leads to an effective development of software products having less complexity. The proposed approach is also helpful in maintaining the legacy of the old system (Majthouab et. al, 2018).

Singh et. al in 2019 conducted a study about the reengineering of software systems (Singh et. al, 2019). It proposed a framework for identifying the need of reengineering, cost estimation of reengineering, and validating the quality improvement reengineered software products. The study made use of agile and scrum methodologies.

Kreedy in 2020 used a simulation-based approach to calculate the risks associated with reengineering projects (Kreedy, 2020). The risks in a project can lead to increase in cost, failure to meet project schedules, and failure to meet project goals. The study used a simulator through Monte Carlo algorithm in Matlab to estimate the risks included in project reengineering.

3.0 Research Methodology

The research work has been carried out in a phased manner. The different phases have been mentioned as follows:

I.Identification of Metrics

Different metrics have been identified that can be used in various phases of the reengineering process. The metrics have been identified considering their usefulness in various areas of software reengineering process.

II.Proposing a Framework

A metric-based framework has been proposed to unearth the cost of the reengineering process. Consequently, a decision about the software product concerning the maintenance/reengineering/retirement can be taken. The proposed framework to be used for deciding if a software product should be maintained or reengineered or retired.

III.Utilization of Metrics

A metric set has been used to calculate whether there is a need to reengineer the software in whole or in partial. The results are indicative of the parts of a software product that may require reengineering.

IV.Validation of Proposed Framework

The proposed framework has been used in different phases of the Rainfall model to make the reengineering process economical and efficient. The Rainfall model has been used as a candidate system to validate the performance and efficiency of the proposed system.

4.0 Case Study and Result Analysis

In this case study source code of software (named TicTac1.java) has been used. The software have 192 statements. This is a simple TicTacToe game program which has been developed without graphics. TicTacToe is a game which is played between two players. One player marks the 'O' sign, and the other player marks the 'X' sign. The player who completes three marks in a row/column/diagonal wins the game.

The only component of the software implements more than one concept, so to increase the cohesion that component should be divided into different modules. To implement this process, above mentioned software needs to be divided into two parts, one part represents internal logic to find win/draw and the second part draws user interface and displays messages on the screen.

I. Phase I: Identification of Candidate System

The software of TicTac1 consists of 1 class of 192 statements and 6 functions (Board (), Check (), main(), Player_win(), PlayerO(), PlayerX()). To improve the maintainability (increase cohesion and reduce coupling) there is a need to break the program into 4 classes and add 2 functions (input_name(), match()). So, 56 statements have to be shifted to the other functions. The new software is called TicTac2.

The ratio of defect age (in years) and software age (in years) ceiling value has been multiplied with lines of code, which are affected by defect, and to total number of lines of code, in order to calculate the defect cost value (Singh and Sood, 2006). For the case candidate, the value of defect age and software age is 1, as the age of software is less than 1 year. Total statements affected by defect are 56 and total statements of the software are 192.

When calculated, the Defect Cost value of the software is 0.29 (56/192). Since the software is changed for the first time, the Fault Cost value of the software is 0.0. Now using the value of Defect Cost and Fault Cost the value of Reengineering Requirement Cost is 5.8. So, there is a requirement for reengineering (Singh and Sood, 2006).

Since the value of Reengineering Requirement Cost is between 3.0 and 6.0, hence there is requirement to calculate Reengineering Requirement Cost of Module (Singh and Sood, 2006) value of each module independently, to find reengineering requirements of each module. After calculating the Reengineering Requirement Cost of Module of each module, results are shown in Table 1.

Table 1: Result Obtained after applying Reengineering Requirement Cost of Module in classes of TicTac2

S. No	**Name**	***Statements***	***Defect Cost***	***Fault Cost***	***Reengineering Requirement Cost***	***Reengineering Requirement Cost of Module***	**Result**
1.	Board	35	.114	---	2.285	5.185	Maintain
2.	Main1	93	.344	---	6.881	9.781	Reengineer
3.	Player	35	.085	---	1.714	4.614	Maintain
4.	Win	49	.102	---	2.040	4.940	Maintain

From Table 1, it can be observed that there is a need to maintain the Board, player and win class, and reengineer Main1 class. Now only the Main1 class is a candidate system for software reengineering.

II. Phase II: Reverse Engineering

In the TicTac1, main () function is connected with 5 other functions. So, its coupling value (Chidamber and Kemerer, 1994) is 5. Since the coupling value of this component is highest, it will work as an important component of the software, i.e., the part from which the reverse

engineering process should be started. Also complexity (Chidamber and Kemerer, 1994) and LOC (Humphrey, 1997) values of main () function are 16 and 74 respectively, and complexity and LOC of Player_win() function are 29 and 45 respectively, which are more than other functions. This makes these two functions good candidates to start with reverse engineering processes.

III. Phase III: Architecture Transformation or Change

The software TicTac1 is implementing all the concepts, Hence, its cohesion is low i.e., 0.33 (Bieman and Kang, 1995). So, to increase the cohesion the software is broken into 4 classes. In the software two more functions are added i.e. input_name() and match(). First function is to read the names of the players and the other is to find the logic of win or draw. Now the cohesion of the software is calculated using tight class cohesion metric (Bieman and Kang, 1995). Cohesion of low_class is 0.67 and cohesion of all high_classes are1.0.

IV.Phase IV: Development of Candidate System

The modifications are performed in the design and are applied through the coding. To prevent the system from behaving abruptly due to the modifications in parts of coding, the critical parts of the system are identified. Function ‘main’ is giving 39 class to other functions (Chidamber and Kemerer, 1994), therefore it should be considered as a point of significance while testing the whole system.

V.Phase V: Integration

During the integration, candidate system (Main1) is combined with player, Board and win classes. The software is tested for complexity (using SourceMonitor) and maintainability using metric Maintainability Index. It is observed that the average complexity of software decreased from 11 to 8.5 and the value of maintainability increased from 96.8 to 102. After reengineering, reengineered software is named TicTac2.

To analyse the difference between TicTac1 and TicTac2 a tool ‘SourceMonitor’ is used. SourceMonitor is a tool that measures and records the source code metrics. After applying the SourceMonitor in TicTac1 and TicTac2, the obtained results are shown in table 2.

Table 2: Result of Tool SourceMonitor on TicTac1 and TicTac2

Software	**Statements**	**% Branches**	**Functions/Methods**	**Average Stmts/ Class**	**Maxmium Complexity**	**Maximum Depth**	**Average Depth**	**Average Complexity**
TicTac 1	192	22.9	6	29.67	29	7	3.65	11
TicTac 2	212	20.8	8	23.25	29	5	2.70	8.5

Column 1 to 9 of Table 2 represents the name of software, number of statements of the table, percentage branches as compared to the whole of the software, number of functions in the software, average statements per function, function with maximum complexity, maximum depth of a function and average complexity of the software respectively.

From Table 2 it can be observed that although LOC of TicTac2 (212) is more as compared to TicTac1(192), maximum depth decreases from 7 to 5, average depth decreases from 3.65 to 2.7 and average complexity decreases from 11 to 8.5.

Pictorial representation of Table 2 is given as Figure 1 and Figure 2 for more clarity.

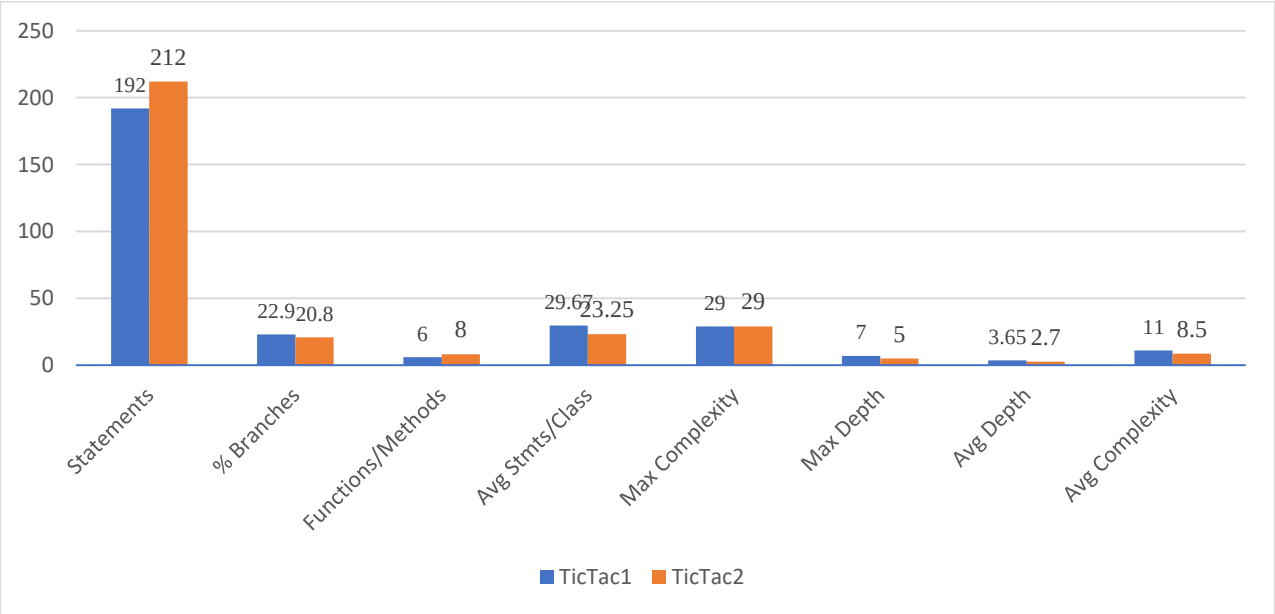

Figure 1: Factor based comparison of TicTac1 and TicTac2

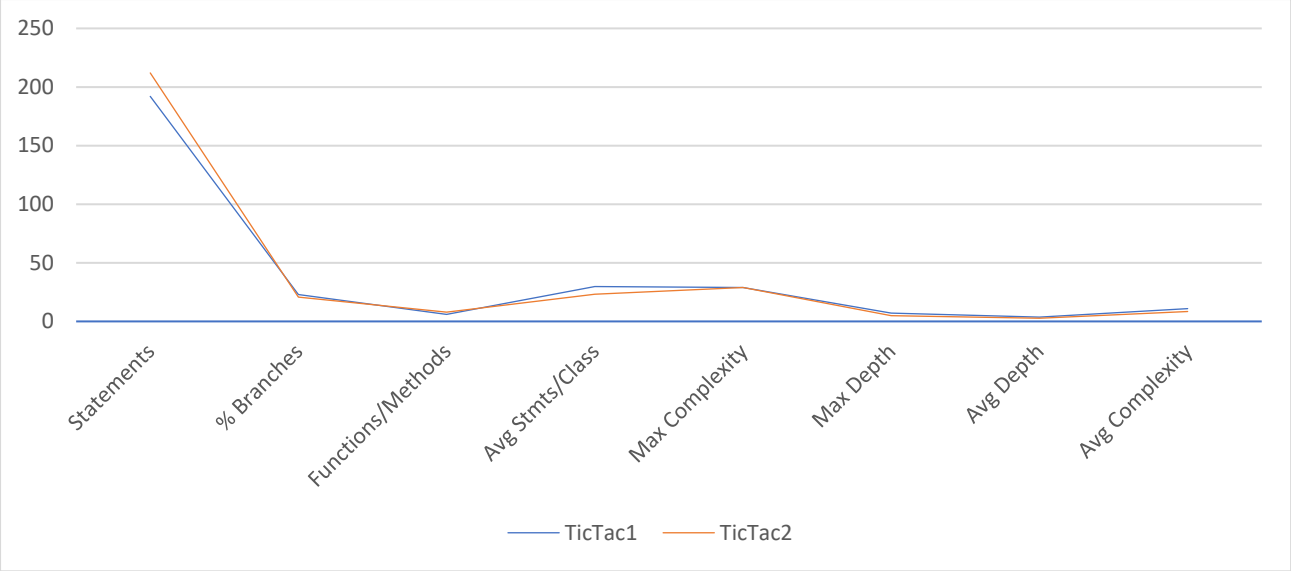

Figure 2: Factor based comparison of TicTac1 and TicTac2

To find the maintainability of TicTac1 and TicTac2 metric Maintainability Index (Oman and Hagemeister, 1994) is used.

Table 3 showing the value of the Maintainability Index of TicTac1 and TicTac2. Column 1 to 7 of the table represents name, number of functions, average Halstead Volume, average cyclomatic complexity, average LOC, average comments lines, and Maintainability Index of the software respectively.

Table 3 Maintainability Index of TicTac1 and TicTac2

Software	**Functions**	**Average Vocabulary**	**Average Cyclomatic Complexity**	**Average LOC**	**Average Comments**	**Maintainability Index**
TicTac1	**6**	**22**	**11**	**32**	**0.16**	**96.8**
TicTac2	**8**	**19.125**	**8.5**	**26.5**	**1.125**	**102**

From Table 3 it is clear that though the number of functions have increased from 6 to 8 in TicTac2 as compared to TicTac1, the average vocabulary has decreased from 22 to 19.125, the average cyclomatic complexity has decreased from 11 to 8.5 and average LOC has decreased from 32 to 26.5. Hence, the maintainability index has increased from 96.8 to 102. So, TicTac2 is more maintainable as compared to TicTac1. The graphical representation of

Table 3 is shown in Figure 3.

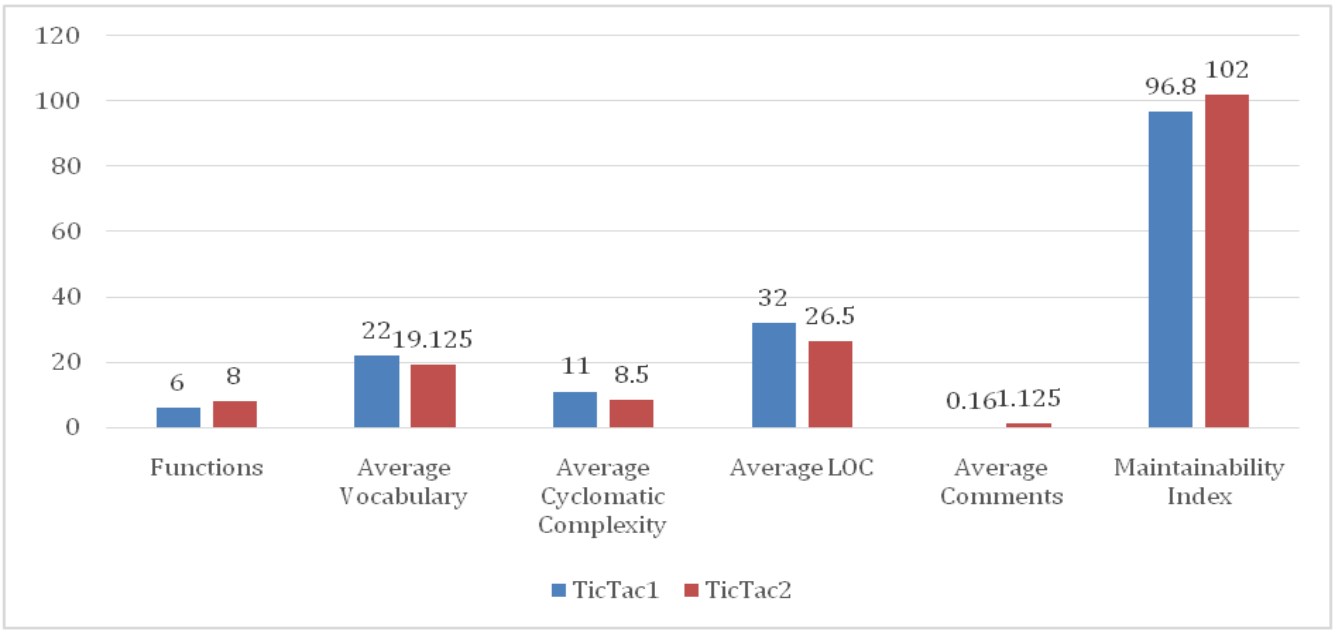

Figure 3: Maintainability Index of TicTac1 and TicTac2

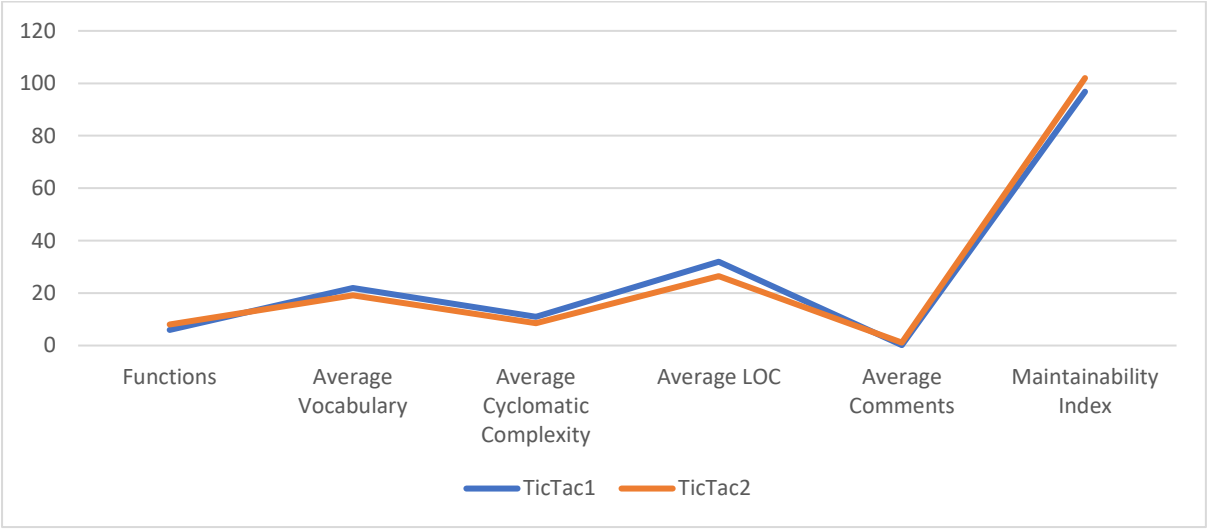

Figure 4: Maintainability Index of TicTac1 and TicTac2

5.0 Conclusion and Recommendations

From the conduct of the study, it is found that when a new version of software is created, a lot of changes occur. These changes can have a major impact on the functionality of the system. It can be helpful in upgrading as well as downgrading of the system. There always exists a risk of making modifications into the system.

The metric assisted approach can be helpful in reducing the effect of risks. Though the risks

cannot be eliminated completely, their impact can be lowered. During the reengineering of software systems, the peril of bringing the system from the old technology to the new platform is always there. That is why; it is recommended through the study that the developers must adopt a metric assisted approach to reengineer a software product. The metric must be selected as per the project requirements and the nature of software.

6.0 References

1. Abowd G., Goal A., Jerding D. F., McCracken M., Moore M., Murdock J. W., Potts C., RugaberS. and Wills L., (1997), MORALE. Mission Oriented Architectural Legacy Evolution, International Conference on Software Maintenance, Italy.
2. Ahrens J. D. and Prywes N. S., (1994), Reengineering the Software Life Cycle and Enabling Technology, Technical Report, Computer, Command and Control Company.
3. Ahrens J. D. and Prywes N. S., (1995), Transition to a Legacy and Reuse Based Software Life Cycle, Computer, 28(10), pp. 27-36.
4. Anitha J.J., Karthika M., and Alagarsamy K., (2012), Risk Analysis for Software Reengineering Process Transformation with XP, International Journal of Engineering Research and Technology, 1 (5), pp. 1-6.
5. Bieman J. M. and Kang B. K., (1995), Cohesion and Reuse in an Object-Oriented System. Proc. ACM Symposium on Software Reusability.
6. Chidamber S. R. and Kemerer C. F., (1994), A Metrics Suite for Object Oriented Design. IEEE Transactions on Software Engineering, 20(6).
7. Chikofsky E. J., and Cross J. H., (1990), Reverse Engineering and Design Recovery: A Taxonaomy, IEEE, pp.15.
8. Duan L., Li X., Li A., and Wang Z., (2013), Application Research on Legacy Software Reengineering in Automated Test System, Journal of Applied Sciences, 13 (8), pp. 1227-1232.
9. Fahmy H. and Holt R. C., (2000), Software Architecture Transformations, Proceedings of the International Conference on Software Maintenance.
10. Humphrey W. S., (1997), Introduction to the Personal Software Process, SEI Series in Software Engineering, Addison Wesley, pp. 22.
11. Konda K. R., (2005), Measuring Defect Removal Accurately, The Enterprise Development Conference, pp. 35.
12. Kreedy A.I., (2020), Estimation of Risk in Software Reengineering Projects, International Journal of Scientific and Research Publications, 10 (6), pp. 799-802.
13. Lehman M., and Belady B., (1985), Program Evolution, Academic Press, London, pp. 9.
14. Leitch R. and Stroulia E., (2003), Understanding the Economics of Refactoring, 5th International Workshop on Economics-Driven Software Engineering Research (EDSER-5): The Search for Value in Engineering Decisions, pp. 44–49.
15. Majthoub M., Qutqut M.H. and Odeh Y., Software Re-engineering: An Overview, proceeding of 8th International Conference on Computer Science and Information Technology (CSIT), 11-12 July 2018, Amman, Jordan.

16. Mishra S. K., Kushwaha D. S. and Misra A. K. (2009), Creating Reusable Software Component from Object-Oriented Legacy System through Reverse Engineering, Journal of Object technology, pp. 133-152.
17. Oman P. and Hagemeister J., (1994), Constructing and Testing of Polynomials Predicting Software Maintainability, Journal of Systems and Software, 24(3), pp. 251-266.
18. Rather M.A. and Bhatnagar V., (2016), Study of Software Development using Software Re-Engineering, international Journal of Engineering Trends and Applications, 3 (2), pp. 52-56.
19. Singh H. and Sood S. (2006), Reengineering Process and Methods: A Study, Proceedings of the conference Innovative Application of IT and Management for Economic Growth, Jalandhar, India, pp. 392-404.
20. Singh H., Sood S., Kaur R., Ratti N. (2007), Metrics Framework for Reengineering Process, Punjab University Research Journal, 57, pp. 251-255.
21. Singh J., Dhindsa, K.S., and Singh J., (2019), Reengineering Framework to Enhance the Performance of Existing Software, International Journal of Advanced Computer Science and Applications, 10 (5), pp. 536-543.
22. Yogesh Hole et al 2019 J. Phys.: Conf. Ser. 1362 012121