Improving the Quality of Software by Refactoring

Gurpreet kaur
Department of Computer Science and Software Engineering
Lovely Professional University,
Phagwara, India
Gurpreetheer70@gmail.com

Balraj Singh*
Department of Computer Science and Software Engineering
Lovely Professional University,
Phagwara, India
Balraj.13075@lpu.co.in

***Abstract*— Software code management has become another key skill required by software architects and software developers. Size of software increases with increase in count of features in software. Code refactoring is process of reducing code maintenance cost. It is achieved by many different techniques like extract, move methods, fields or classes in code. In this research we focused on improving the maintainability of the code by looking into the different refactoring techniques and improving upon them.**

We proposed an algorithm to improve the refactoring process which results in higher maintainability. To look into the validity of our proposed algorithm, we have used Junit and ref-finder to analyse the code and generate the result metrics. We have observed the effectiveness of our work by comparing the different code maintainability indexes generated by the tool. In our research we have examined four releases of the software project for code refactoring and maintainability. Adding some extra features and using enhanced refactoring techniques measuring the code metrics and comparing the results of current releases with the previous releases.

***Keywords*— *Code refactoring, bad smells, refactoring process, software metrics, software quality attributes*.**

I. INTRODUCTION

Refactoring approach is used to refine the internal structure (part) of code without damaging the external activities of the software [21]. Refactoring approach is used to decreases the complexity of the software by fixing errors or appending new features. Refactoring also improves the performance of the software. Refactoring is also involved in reengineering process to enhance the quality of the software. The aim of the refactoring approach is to maintain the code of software and make it healthier.

The process of the transformation of the source code can be done by the refactoring. The achieved transformation through refactoring makes the software easy to understand without changing the observable behavior. The different refactoring methods that are used in the code at right place can be beneficial for the incremental improvement in the software quality [20] [21]. To remove or lower the defects for the improvement of the software quality, refactoring is done manually. The main aim of the refactoring is alteration of the code safely to enhance the quality. Refactoring techniques are utilized to refine the code. Different refactoring techniques are created for implementing with suitable quality attributes and metrics. The cost of software maintainability can be decreased for long time by using refactoring on the software code. The existing software problems can be removed by enhancing the software code with the help of refactoring. The software can be improved by manipulating the code. The action of refactoring can modify the internal activities with the purpose of accepting its processes. In the process of software development, the software system is implements first and then the code for implementation purpose is written. Refactoring has both positive and negative effects on the quality of the software. Factors such as high power consumption extend execution time, additional memory used were also examined. The refactoring process upgrades the software quality by adding new features to the code and by removing the bad smells.

Bad smells is used to indicate the poor design [8]. Some bad smells like duplicate code, long method, long class, long parameter list, switch statements, message changing, too much communication between objects etc. Bad smells are mostly easy-to-spot signs in the code.

RQ1. Effect of refactoring on low maintenance code?
RQ2. Which attributes of the code are affected most by refactoring?
RQ3. Impact of code re-factoring on future releases in terms of ease of adding new features and removing a feature with minimal changes.

II. RELATED WORK

Software or code refactoring has become a major area of research these days. The following research questions are formulated to evaluate categories and summarize the findings of the accumulated software refactoring literature:

I. Kádár *et.al* in 2016 [1]. In this paper the author proposed the future inspection of code refactoring in practice by producing a necessary open dataset of source code metrics and utilized refactoring through various releases of 7 open source system. The author explored the quality attribute of the refined source code classes and the effectiveness of source code metric upgrade by refactoring techniques [1] [16]. The author evaluated the correlation between maintainability and refactoring methods and also examined how source code metric can be done by refactoring affect. The author proposed the dataset including refactor data and more than 50 types of

source code metrics for 37 releases of 7 open source system at the class and procedure level.

Study from Istvan Kadar *et.al* in 2016 [2]. In this paper the authors manually performed the refinement of the code to obtain the dataset. They evaluated the dataset to find whether the refactor code operations with refactoring activities and law maintainability used by the authors relates to the internal quality or not. For this method, they studied the maintainability values in the datasets by using Mann-Whitney U test on different set of data formed by the particular item whether they were affected by the refactoring methods [2] [13]. The investigation showed that the average maintainability of refactor data is much lower. The manually formalized refactoring dataset included only the approved data which was obtained from original dataset.

Study from Gabriele Bavota *et.al* in 2015 [3]. In this paper the authors performed study on three java open source software system to evaluate the relationship between refactoring and code quality. The research has organized 63 releases of three java system software and involves the manual survey of 15,008 refactoring operations and 5478 smells. The refactoring performed on those classes which were affected by the smells was analyzed to be 40% and only 7% smells were actually removed. In this paper the quantitative method was used to perform refactoring techniques [3] [7]. They also used coupling metrics to measure the effect of refactoring and selected the quantative method to choose relevant refactoring type. In this paper they measured the complexity, clone metrics and size of the refactor data.

Study from anshu rani *et.al* in 2012 [7]. In this paper the authors discussed some refactoring techniques, tools and some features for code refactoring. Basically refactoring is used to enhance the internal quality, maintainability and reliability without affecting external structure. The author proposed some steps to perform refactoring on code like identifying the code where refactoring should be applied or determining the refactoring methods which can be used for particular place, assurance about maintaining behavior, applying refactoring technique and accessing the results of refactoring code[7] [13].

Study from Anam shahjahan *et.al* in 2015 [5]. In this paper the researchers proposed a new study to enhance the features of the code by using graph theory techniques. Refactoring is a procedure of enhancing the quality of code without changing its internal structure and external part. They used hypothesis techniques to correlate the results that produced. Response time is also got improved in this study. Analyzability, changeability, time behavior and resource utilization are main four qualities attributes that are used to improve code quality.

Study from Yoshio kataoka *et.al* in 2002 [15]. In this paper the authors proposed a quantative assessment method to calculate the improved maintainability results of code refactoring. The author concentrated on the coupling metrics to assess the effect of refactoring on code. In this paper the author compared the coupling before using refactoring methods and after using refactoring techniques to improve the quality and assess the maintainability improvement. In this paper the author used three coupling metrics and combined these three coupling metrics to evaluate the code using different code refactoring methods [15] [25].

Study from Michael Wahler *et.al* in 2016[36]. In this paper the author reports on a case study in which magnetic researchers were consulted by software engineers in refactoring their simulation software. The stakeholders of the research project considered the software to be un-maintainable as it had reached to a size of 30 kilo line of code of Java. The case study states the procedure of refactoring the system under the guidance of a software engineer with results supported by static analysis and software metrics. It tells how software engineers evaluated and selected refactoring to incorporate to the system using their precise judgment with input from static analysis tools and discusses the outcomes of refactoring as evaluated by code owners and reported via static analysis metrics.

They presented a case study on refactoring a design tool for magnetic components so that its maintainability could be increased. In order to prioritize the maintenance tasks, they combined the results from automatic code analyses with the subjective assessment of the original developer. The combined assessment was also used to measure the success of their refactoring activities. The results were encouraging. The number of potential issues found by Find Bugs was reduced by 23 % and around 82% of amount of duplicate lines of code was reduced.

Study from Chaitanya Kulkarni *et.al* in 2016[37]. In this paper the author aims mainly towards the possibility of detecting a refactoring code and to find out whether the code clone can be securely refactored or not. Three methods were applied: Nesting Structure Mapping, Statement Mapping and Precondition Examination. They applied some techniques like Pull-Up Method and Push-Down Method in order to refactor the code. In their approach, they tried to find the refactorable code by using methods and also refactored the code so as to remove the problem of code cloning.

Study from Minas F. Zibran *et.al* in 2015[38]. In this paper author tells about characteristics of clones can be understood by clone analysis and visualization. A number of studies have analyzed clones and their evolution while a numerous techniques have also been proposed in order to visualize the clones that aid in clone analysis. However, clone analyses and visualizations with respect to inheritance hierarchy and call graphs have remained ignored so far. In this research paper, the author argued that such analyses and visualizations with respect to the inheritance hierarchy and call graphs are necessary to help in dealing with clones for refactoring.

TABLE I: Summary Of Refactoring Techniques

Authors	Case Study	Internal Measures	External Measures	Refactoring
Kataoka et al.[29]	A C++ program	Coupling	Maintainability	Extract Method and Extract Class
Stroulia and Kapoor et al.[27]	Academic	Size and coupling	Design extendibility	Extract Super class, Extract abstract class
Leitch and Stroulia et al.[31]	Academic and commercial	Code size, number of procedures	Maintenance effort and costs	Extract Method, and Move Method
Tahvildari and Kontogiannis et al.[32]	Four open-source applications	coupling, cohesion, inheritance and complexity	Maintainability	Code Transformations
Bois et al.[30]	open source software	cohesion and coupling	-	Extract Method, Move Method, Replace Method w Method Object, Replace Data Value w Object, and Extract Class
Moser et al.[35]	A project in industrial environment	LOC, CK measures, Effort (hour),	Productivity (LOC)	
Alshayeb et al.[34]	Three small Open-source projects	CK measures, LOC, FOUT	adaptability, maintainability, understandability, reusability, and testability	Extract Class, Encapsulate Field, Extract Subclass, Move Class, Extract Method, Replace Temp with Query, and Extract Subclass
Sahraoui et al.[26]	A C++ program	Inheritance and coupling measures	Fault-proneness	Extract Super class, Extract Subclasses, Extract Aggregate Classes
Tahvildari et al.[32]	A project in industrial environment and open library; both written in C.	Halstead's efforts, LOC, and number of Comment lines per module	Maintainability and performance	Design patterns
Noble Kumari et. al[14]	Open source code	Coupling, Cohesion and inheritance	adaptability, maintainability, understandability, reusability, testability, fault Proneness, stability and completeness	Wrap Return value, Safe Delete and Replace Constructor with Builder method

III. RESEARCH APPROEACH

Refactoring is a technique which is used to enhance the internal quality of the software without changing the external behavior of the software. Internal quality attributes are used as a software metrics and software metric is used to evaluate the maintainability of the software. In our research we will evaluate project for code refactoring and maintainability of code taking four releases of the project. Ref-Finder will be used as tool to extract code refactoring differences between releases of project. To measure code metrics in each release we will use halstead tool that is easily used to measure code metrics and refactoring problems in the code.

Following are the steps of proposed methodology:

1. Gather source code from previous dataset.
2. Scan each release individually for code metrics
3. Measure code metrics
4. Apply the enhanced re-factoring techniques
5. Measure code metrics again
6. Compare result with existing techniques

A. Algorithm: Proposed algorithm for enhanced refactoring technique which is mentioned in step 4 of methodology.

1) Scan Junit releases 4.9.1, 4.10, 4.11 and 4.12
2) Find following in code scan:
   a) Impact of code re-factoring on future releases in terms of ease of adding new features and removing a feature with minimal changes
   b) Effect of refactoring on low maintenance code.
   c) Which attributes of the code are affected most by refactoring?
3) Create list of refactoring candidate classes
4) For each candidate
   a) Scan class to find:
      i. Generate class flow for methods
      ii. Variables have getter /setter methods
      iii. Methods have flow which cannot be further divided
      iv. Scan code fragments to find similar code
      v. If similar code exists in different methods then
         I. Flag class as refactoring
      vi. Scan methods for variables used
   b) Assign class score for refactored code in variables and methods
5) For each class having score >8 generate list for suggestion of lists for missing refactoring.

Here is flowchart depicting methodology to be followed for research:

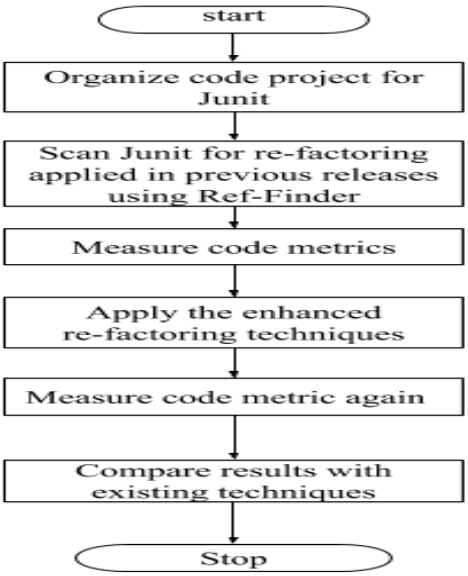

Fig1. Proposed methodology

To look into the source code refactoring practically, we worked on the source code estimations and associated refactoring techniques. We assess the relationship between the numbers of refactorings techniques impacting the product. We inspected the current techniques of refactoring a source code through the quantified metric values and proposed an enhanced algorithm which performs better in refactoring an existing code.

B. Data Construction

The dataset contains information release version of Junit open source java framework accessible in GitHub which gives details about projects. This project was chosen for our research reason due to the adequate number of releases adaptation and the measure of code between two adjacent releases. We examine 3 to 4 arrival of Junit system. For each release version of Junit system, class and methods level measurements and the number of refactoring assembled by refactoring systems. Table II gives the aggregate number of classes, methods and refactoring procedures.

TABLE II**.** Total Number Of Classes, Methods And Refactoring

System	**No. of classes**	**No. 0f Methods**	**Refactoring**
Junit existing	1,267	4,124	553
Junit refactored	1,267	4,124	200

We played out a relationship examination on the RMI estimations of the classes and the amount of refactorings affecting these classes. We took the RMI values from releases, and the amount of refactorings from releases. We assessed whether low quality classes got refactored more truly than various classes or not. We figured the differences of the metric values between the resulting releases. A significant part of the

time negative differentiations mean a change, as lower metric qualities are better.

IV. RESULTS

In this section we compress the appraisal consequences of the gathered refactoring dataset with respect to software maintainability. In the first place, we describe the results of the investigation on the maintainability of refactored classes to answer RQ1. A while later, we introduce the findings on the impact of refactorings on source code measurements to answer RQ2. Applying RQ3 helped in upgrading the RMI index there by reducing refactoring requirement on build on build basis in Junit dataset.

The graph below shows ratio of change in metrics after applying refactoring derived by 3 questions that has were defined in initial objectives of our research. The refactoring techniques move method and mode, move field, extract class are applied to Junit releases 4.12, 4.11 and 5.1 releases to compare the effectiveness of our approach for refactoring.

Applying questions RQ1 & RQ2 helped in maintaining the coupling and separation of concerns in classes. Applying RQ3 helped in upgrading the RMI index there by reducing refactoring requirement on build basis in Junit dataset.

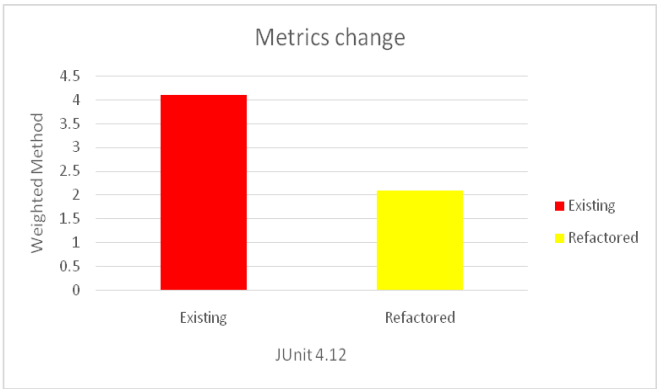

Fig 2.Metric change

A. Nested block depth: Nested block depth helps in identifying that if a method or class is serving more than one purpose that would keep on adding LOC to class/method release by release ultimately making it unmanageable after some time. Lower the NBD is more manageable class. Nested block depth increases complexity of code and thus adds to maintainability of the code. Simplifying nested blocks and replacing it with inherited classes helps in maintaining simplifying it.

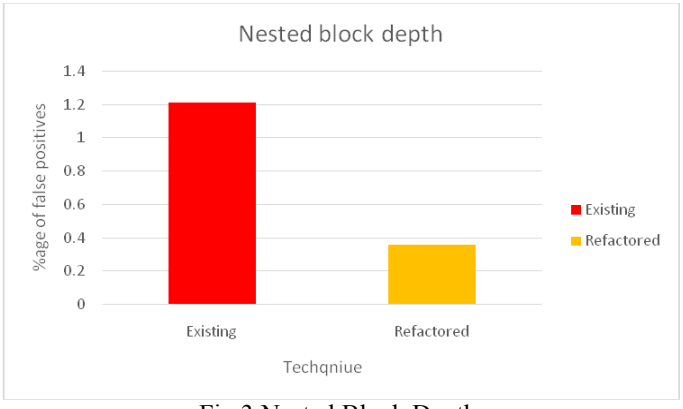

Fig 3.Nested Block Depth

B. Number of parameters: : NOP increase with increase in desired functions in a method. The more parameters are added complexity of method would increase with NOP. Thus lower NOP helps in maintaining code maintainability. NOP increases with increase in complexity of methods, applying future release method helps in reducing method parameters and thus reducing NOP.

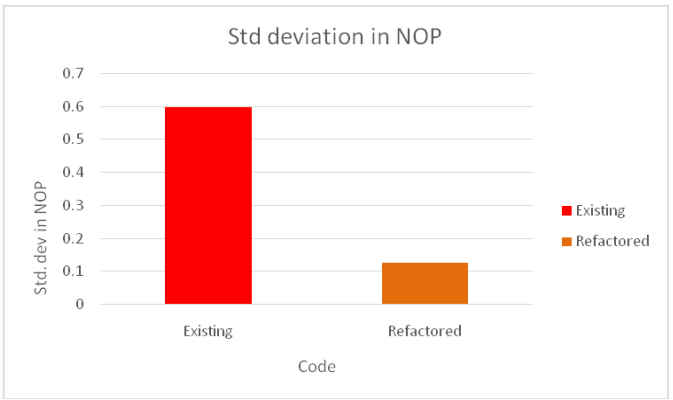

Fig 4.Number of parameter

C. No. Of classes: No. Of classes in a code management defined how separation of function in classes. Number of classes increase as we implement refactoring. Applying futuristic approach increase need of loose coupling in classes thus increasing number of classes

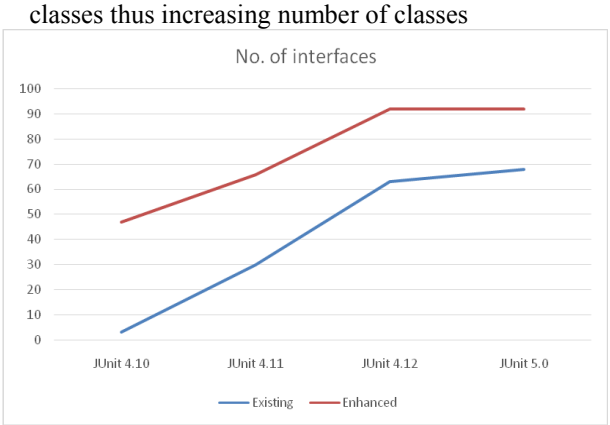

Fig 5. Number of classes

TABLE IV. Comparison Between The Existing And Refactored Parameters:

Parameters	Refactored	Existing
Weighted method as per class (complexity)	0.2	0.4
Nested depth block	0.3	1.2
Number of parameters	0.1	0.8

The comparison table defines the values or results of refactored and existing is based on the above mentioned graphs. The results shows that the proposed technique is better as compare to existing techniques code maintainability index.

D. Effect of Refactoring on Source Code Metrics

According to the process, we first calculate the metric value differences for every class between the adjacent releases [1]. At that point, we gathered these metric distinction values into two gatherings: in the primary gathering we put the metric contrasts of classes touched by at least one refactoring, and in the second gathering the metric contrasts of non-refactored classes.

E. Software Metrics

Software metrics are also used as the internal quality attributes. Software metric is better occurrence of measuring the quality of software. Measuring the complexity of the system is the common procedure to estimate the maintainability of the software.

We found that size (TLLOC, TNOS), coupling (RFC, NOI), clone (CI), complexity (WMC) and comment (TCLOC) related measurements diminish the most in refactored classes.

Table III. Metric Improvement

Enhanced							
System name	**CI**	**WMC**	**NOI**	**RFC**	**TCLOC**	**TLLOC**	**TNOS**
JUnit 4.10	0.728	0.042	0.17	N/A	0.012	0.101	0.113
JUnit 4.11	0.586	0.025	0.0987	N/A	0.0098	0.08654	0.0875
JUnit 4.12	0.5264	0.018	0.0654	N/A	0.0086	0.07754	0.0775
JUnit 5.0	0.444	0.008	0.0274	N/A	0.0076	0.07208	0.062
Existing							
JUnit 4.10	0.8736	0.0504	0.204	N/A	0.0144	0.1212	0.1356
JUnit 4.11	0.7032	0.03	0.11844	N/A	0.01176	0.103848	0.105
JUnit 4.12	0.63168	0.0216	0.07848	N/A	0.01032	0.093048	0.093
JUnit 5.0	0.5328	0.0096	0.03288	N/A	0.00912	0.086496	0.0744

With respect to volumes of the distinctions, we can say that for these measurements the normal chances of is 4-9 times higher in the classes influenced by refactorings than in the non-refactored classes.

Table III. Basically defines the comparison between existing and enhanced metric improvement. Which defines how proposed technique is better than existing one. These values are generated after applying the proposed algorithm and find the improved metrics. The main objective behind this research is to extract Junit code with the help of ref-finder tool and calculate the metric for the code and after calculating we applied proposed technique and applied refactoring techniques on it and refactored the code. again we calculate the metric for refactored code and compare it with the existing technique. which describes that the enhanced technique is better than the existing technique.

V. Conclusion

The main goal of this research is to addresses the gaps in practical and theoretical code re-factoring techniques. We release connections between re-factoring, code metrics and bugs that are discovered during code reviews and analysis cycles.

We evaluate the set of steps that can be followed to ensure low code maintenance and enhanced reliability also, minimizing efforts required for re-factoring during development of software. In our research we analyze software project for code refactoring and maintainability of code taking releases for the project. Ref-Finder was used as tool to extract code refactoring and use to compare the results of previous releases and new releases and analyze the present releases. To measure the code parameters we use code maintainability index. To measure code metrics in each release we use Hal-

stead as plug-in that is easily used to measure code metrics and refactoring problems in the code. Adding some extra features and using enhanced refactoring techniques measuring the code metrics and comparing the results of current releases with the previous releases.

As per the result section proposed technique out performs the existing techniques in terms of RMI. Maintainability index of software code provides a way to ensure that code is manageable and addition/changes in features of software is less prone to risk as compared to code that requires high refactoring. Proposed technique of refactoring has reduced build on build requirement of refactoring thus making it a better approach for refactoring. The current proposed work is limited to medium scale projects and maintainability index is also developed for medium scale maintainability. Further applications easily propose work can be done on large scale project to take it into effectiveness in the context of maintainability index.

References

[1] I. Kádár and P. Heged, "A Code Refactoring Datasetand Its Assessment Regarding Software Maintainability," *IEEE 23rd international conference on software Analysis* , 2016.

[2] I. Kádár and P. Heged, "A manually validated Code Refactoring Dataset and Its Assessment Regarding Software Maintainability," *IEEE 23rd international conference on software Analysis* , 2016.

[3] G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba, "An experimental investigation on the innate relationship between quality and refactoring ," *J.syst.Softw.,* vol.107.pp.1-14, 2015.

[4] I. Verebi, "A model-based approach to software refactoring," *2015 IEEE Int. Conf. Softw. Maint. Evol.*, pp. 606–609, 2015.

[5] A. Shahjahan, "Impact of Refactoring on Code Quality by using Graph Theory : An Empirical Evaluation," pp. 595–600, 2015.

[6] K. O. Elish and M. Alshayeb, "Using software quality attributes to classify refactoring to patterns",*J.Soft.,* vol. 7, no. 2, pp. 408–419, 2012.

[7] A. Rani and H. Kaur, "Refactoring Methods and Tools", *Int j.Adv. Res. Compt. Sci. Soft.Eng.* vol. 2, no. 12, pp. 117–128, 2012.

[8] K. O. Elish and M. Alshayeb, "Using software quality attributes to classify refactoring to patterns",*J.Soft.,* vol. 7, no. 2, pp. 408–419, 2012.

[9] M.Fowler, K. Beck, J. Brant, W.Opdyke and D. Roberts, "Refactoring Improving The Design of Existing Code",*Addison Wesley*.

[10] A. Moeini, V. Rafe, and F. Mahdian, "An approach to refactoring legacy systems", *ICACTE 2010-2013rd Int.Conf. Adv. Comput. Theory. Engg. Proc.*, vol. 5-8,2010.

[11] K. O. Elish and M. Alshayeb, "Investigating the effect of refactoring on software testing effort", *Proc. - Asia-Pacific Softw. Eng. Conf. APSEC*, pp. 29–34, 2009.

[12] Bart D Bios and Jan Verelst, "Refactoring- improving coupling and cohesion of existing code", *11th working conference on reverse engineering, 2004.*

[13] Tom Mens and Tom Tourwe, "A Survey of Software Refactoring", *", IEEE transaction on software engineering, VOL. 30, NO. 2, 2004.*

[14] Noble kumari and Anju Saha, "Effect of refactoring on software quality", *Proc. Conf. Softw. Maint., pp. 37–46, 2014*

[15] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya, "A quantitative evaluation of maintainability enhancement by refactoring," *Softw. Maintenance, 2002. Proceedings. Int. Conf., pp. 576–585, 2002.*

[16] P. Oman and J. Hagemeister, "Metrics for assessing a software system's maintainability," *Proc. Conf. Softw. Maint. 1992, pp. 337–344, 1992.*

[17] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object-oriented design," *IEEE Transactions on Software Engineering, 20(6):476–493, 1994.*

[18] B. Beizer, "Software Testing Techniques," *Van Nostrand Reinhold, New York, **NY,** 1990*

[19] H. Zuse, "Software Complexity Measures and Methods," *Walter de Gruyter & Co., New York, NY, 1991.*

[20] Swarnendu Biswas and Rajiv Mal, "An approach to software engineering," 2009.

[21] P. Jalote, "A Concise Introduction to Software Engineering," *Addison-Wesley,2002.*

[22] M.Fowler, K. Beck, J. Brant, W.Opdyke and D. Roberts, "Refactoring: Improving the Designof Existing Code," *Addison wesley,1999.*

[23] H.A. Sahraoui, R. Godin, T. Miceli, "‖Can Metrics Help To Bridge The Gap Between The Improvement of OO Design Quality And its Automation?", ‖ In: Proc. International Conference on Software Maintenance*, pp. 154–162, 2000*.

[24] E. Stroulia, R.V. Kapoor, "Metrics of Refactoring-Based Development: an Experience Report," *In The seventh International Conference on Object-Oriented Information Systems, pp. 113–122, 2001.*

[25] S. Demeyer, "Maintainability versus Performance: What's the Effect of Introducing Polymorphism? technical report, Lab. On Reengineering," *Universiteit Antwerpen, Belgium, 2002.*

[26] Y. Kataoka, T. Imai, H. Andou, T. Fukaya, "A Quantitative Evaluation of Maintainability Enhancement by Refactoring‖," *Proceedings of the International Conference on Software Maintenance (ICSM.02), pp. 576–585, 2002.*

[27] B.D. Bois, T. Mens, "‖Describing the Impact of Refactoring on Internal Program Quality‖," *In Proceedings of the International Workshop on Evolution of Large-scale Industrial Software Applications (ELISA), Amsterdam, The Netherlands, pp. 37–48, 2003.*

[28] R. Leitch, E. Stroulia, "‖Assessing the Maintainability Benefits of Design Restructuring Using Dependency Analysis‖," *Ninth International Software Metrics Symposium (METRICS'03), pp. 309–322.*

[29] L. Tahvildari, K. Kontogiannis, "‖Improving Design Quality Using Meta-Pattern Transformations: A Metric-Based Approach‖," *J. Software Maintenance. Evolution: Research and Practice, 16 (4-5), (2004) pp. 331–361.*

[30] L. Tahvildari, K. Kontogiannis, J. Mylopoulos, "‖Quality-Driven Software Re-Engineering‖," *Journal of Systems and Software, Special Issue on: Software Architecture - Engineering Quality Attributes, 66(3), (2003) pp. 225-239.*

[31] M. Alshayeb, "‖Empirical Investigation of Refactoring Effect on Software Quality‖," *Information and Software Technology, 51 (9), (2009) pp. 1319–1326.*

[32] R. Moser, P. Abrahamsson, W. Pedrycz, A. Sillitti, G. Succi, "‖A Case Study on the Impact of Refactoring on Quality and Productivity in an Agile Team In Balancing Agility and Formalism in Software Engineering, *Bertrand Meyer, Jerzy R. Nawrocki, and Bartosz Walter (Eds.). Lecture Notes In Computer Science, (5082). Springer-Verlag, Berlin, Heidelberg*, pp. 252-266, 2008.

[33] M. Wahler, U. Drofenik, and W. Snipes, "Improving Code Maintainability : A Case Study on the Impact of Refactoring", 2016.

[34] C. Kulkarni, A Qualitative Approach for Refactoring of Code Clone Opportunities Using Graph and Tree methods, 2016.

[35] M. F. Zibran, "Analysis and Visualization for Clone Refactoring," *pp. 47–48, 2015.*

[36] A. Vasileva and D. Schmedding, "How to Improve Code Quality by Measurement and Refactoring,," *2016.*

[37] S. H. Kannangara and W. M. J. I. Wijayanayake, "Impact of Refactoring on External Code Quality Improvement : An Empirical Evaluation," pp. 60–67, 2013.

[38] G. P. Krishnan and N. Tsantalis, "Unification and Refactoring of Clones," pp. 104–113, 2014.