iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 9 **Issue:** XI **Month of publication:** November 2021

DOI: https://doi.org/10.22214/ijraset.2021.39007

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.429
Volume 9 Issue XI Nov 2021- Available at www.ijraset.com

Empirical Analysis of Re-Factoring Techniques

Dendi Naishika Reddy
VIT Vellore; India

Abstract: ***The process or technique of Code Re-factoring is restructuring the existing source code by making changes in factoring without any changes in external behaviour. The main intention of re-factoring is to improve non-functional attributes of the software. The advantages include improving the code readability and reducing the complexity of any given source code, and these can overall enhance code maintainability and produce a much more elaborated internal architecture or object-oriented model to boost the extensibility of the code. The effect that re-factoring has on any software project is analysable and customisable. But, before customising the factoring techniques, it is essential to have a complete knowledge of all possible re-factoring techniques, and all its possible effects. Our main focus will be on few main re-factoring techniques like Red-Green re-factoring, preparatory re-factoring, Abstraction re-factoring, composing methods re-factoring etc. Every software project has both internal and external attributes, that highly influence the software's maintainability, reusability, understandability, flexibility, testability, extensibility, reliability, efficiency, modularity, complexity and composition. The research mainly focuses on the effect of re-factoring on them. Study of researched data will give us comparative analysis, pointing out both the positive and negative impacts, re-factoring can have. Overall, the project aims to perform an empirical study to find out the impacts of re-factoring techniques. The research aims to explore the change in the quality of the code after re-factoring. Improvement, decrement and stability are analysed. Study is also done to find the possibilities of applying more than one re-factoring techniques, independently or in an aggregation.***

Keywords: maintainability; extensibility; reliability; modularity

I. INTRODUCTION

To study the effects and methods of re-factoring, fundamental parts of the object-oriented code sample were refactored. The S.O.L.I.D principles were taken as basis to figure out the possibilities of re-factoring an object-oriented code. The results of basic empirical study revealed that re-factoring has increased the maintainability. The performance of the complete program is then analysed to find out about the dynamic binding which has a major impact on the execution time of the program. This study resulted in growth in the execution time.

The motivation to work on this idea was solely based on simplifying the given object-oriented code. And to do so, we are generally focusing on re-factoring techniques to simply the code. This technique was chosen due to its ability to affect the results by changing the factors in the code. The entire code need not be changed. Improving readability of a code, and reducing complexity are few of the many advantages of re-factoring. However, these can be considered as advantages only when maintainability is also improved, and extensibility is enhanced giving an expressive internal architecture. Given, our main focus being the simplification of the given code, hence the advantages of re-factoring makes we combine the re-factoring with our code, to optimize it and make it simpler for the user. The importance that re-factoring has on real life applications and automation.

The problem statements that would be covered in this research would include- what re-factoring scenarios were to be considered and checking whether there is an opportunity to refactor, what are the steps involved in re-factoring any code of a software project, what quality attributes, measures and approaches were to be considered and empirical techniques which can be used to investigate the effect of re-factoring on software code quality. Datasets to investigate connection between re-factoring techniques and software code quality, what are the software internal and external attributes and effect of re-factoring on them, And overall effect of re-factoring on software projects as a whole.

II. METHODOLOGY

Re-factoring techniques have different effects in different scenarios. Thus, we have selected the evolutionary model. Each time a technique is applied on a code, its effects are analysed, the code is tested and validated based on its quality, and either the technique is implemented or discarded based on the impact it has on the quality of the code. As shown in Figure 1, below.

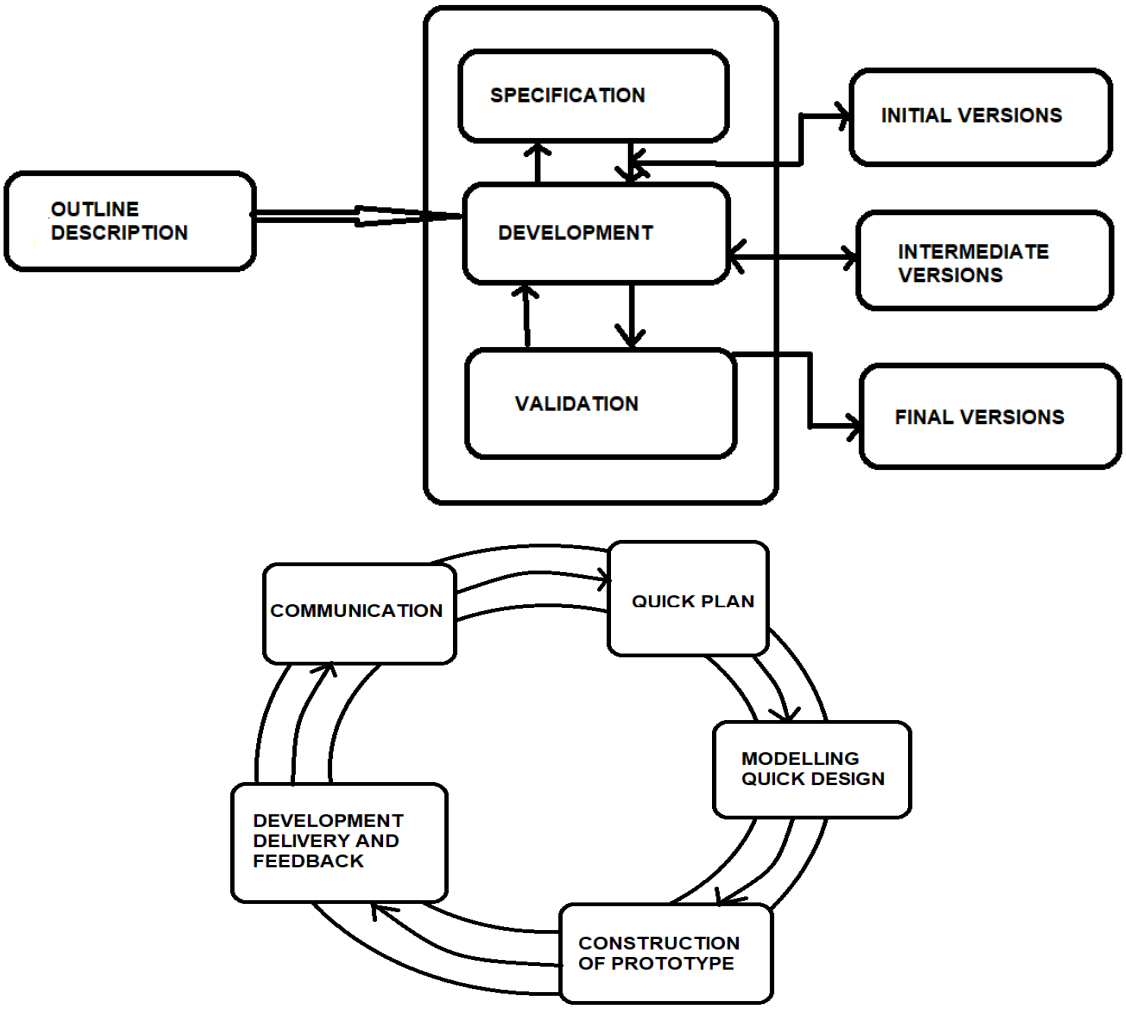


Figure 1: Selected process model- evolutionary model.

1) Outline Description

a) Initial step is to draft a code, specifically an object-oriented code, thus ensuring modularity.

b) Selecting the re-factoring techniques to be implemented and tested.

2) Specification

a) Checking for opportunities to refactor the code.

b) Identifying different ways to implement the re-factoring techniques.

c) Software tools required to refactor, exploring and acquiring them.

3) Development

a) Using the software tools and platforms, apply the re-factoring techniques.

b) Measure the external and internal quality attributes (also using software tools)

4) Validation

a) Compare the measured values of quality attributes before and after applying the re-factoring techniques.

b) Obtain graphical data if required

c) Report the outcomes

d) If a positive change is found, include technique in final versions, else discard and apply the next re-factoring technique. Follow the same model for all the techniques chosen.

So our proposed model can be redesigned as:

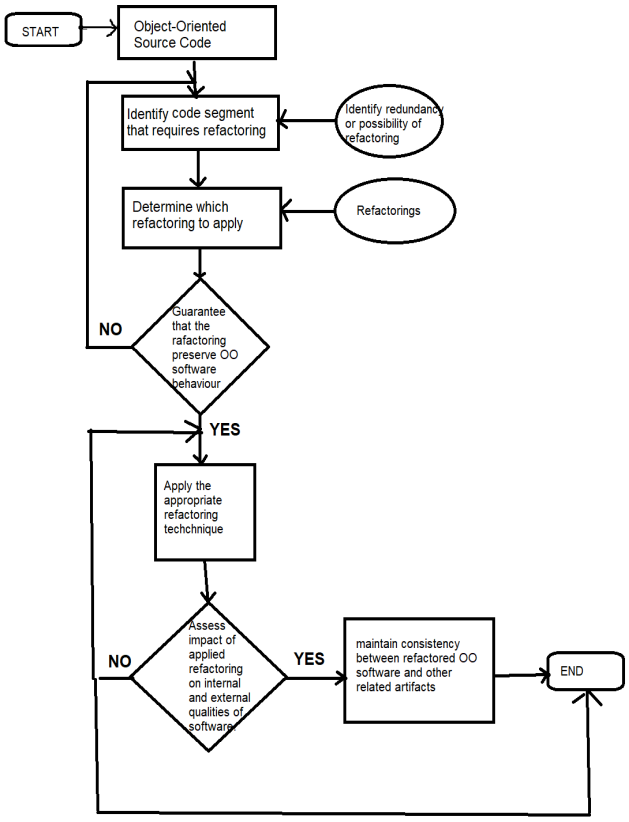


Fig 2: Proposed model for re-factoring object-oriented code.

Each time a re-factoring technique is picked amongst the known techniques like - Red-Green re-factoring, preparatory re-factoring, Abstraction re-factoring, composing methods re-factoring etc, a quick implementation is performed. Assuming the obtained code as a prototype, it is validated against the required quality standards of the code. If the quality is found to be enhanced, then the re-factoring technique is incorporated, else, the prototype is discarded and a new technique is selected and testing in the same way.
As it can be seen that the process of identifying the possibility of re-factoring, requires a very particular analysis of the entire code, manually finding the possibility becomes time consuming and ineffective. That is the reason why it is generally automated. There are a number of re-factoring tools that have are automized. They scan the entire code, find the segments for re-factoring and also apply the re-factoring techniques. Thus, the manual effort and time consumption is reduced and efficiency is increased.

A. Theoretical Aspect And Case-Study Analysis

Theory behind some of the re-factoring techniques:

1) *Extract Subclass:* A particular subclass having functions that are a subset of the functions of super class that are used by the program only in few instances. A case study conducted on this technique reveals that super-class has functions and methods that are implemented only in few rare cases. Hence these functions are included in a subclass and called by the program only when required. This method is used to subdivide main-class or super-classes into sub classes. These subclasses have only minimal functionalities, and rare uses. The benefit of this technique is that it efficiently creates a subclass. In instances where the super-class has many such functionalities, then more than one of such subclasses are created.
2) *Move Method:* The technique is used to move a functionality from one class to another where it is mostly used rather than the one it is already present in. This technique creates a function in a new class, where it is mostly used, and later move the code for that function from a class where the function is already present but not majorly used. The function in the parent class can either be referred by the new function or it can completely be deleted. Thus, this makes the classes more internally coherent. Also, study reveals that this technique reduces the dependency between classes.

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.429
Volume 9 Issue XI Nov 2021- Available at www.ijraset.com

3) *Parameterize Method:* Different methods use parameters with different names but representing the same data. This method globalises the parameters reducing redundancy. Many identical groups of parameters are often encountered in multiple methods. This causes redundancy. By consolidating parameters in a single class, this redundancy can be reduced. The benefits of this technique are that instead of a clumsy data set, a single object/parameter with a comprehensible name can be used by a greater number of methods. Similarly, each of the 10 well known re-factoring techniques has a unique approach and case studies and theoretical knowledge of these techniques reveals their benefits. However, each technique has its disadvantages as well, which can only be analysed and identified by the empirical study, which is the major scope of this work.

III. RESULTS

To find the impacts of re-factoring on any software code quality, its impacts on internal and external qualities is considered. Taking in consideration a set of these qualities, empirical study of the outcomes will give a possible conclusion on the impact re-factoring has. To do so, we have used automized re-factoring tools such as "Re-factoring Essentials" (Open-source free visual studio extension for C#, VB.NET code re-factoring), Visual studio code re-factoring tool.

A lot of research has already been done to find out the impacts of re-factoring on both internal and external attributes of software code. After studying the various reports, the main aim of this empirical study is to either validate the findings or come up with new conclusions. The study also aims to determine the worth of expenses and time spent on re-factoring.

In order to reach to a conclusion, our study was majorly conducted using two measures namely internal and external measures.

Coming to the re-factoring techniques, there are essentially 10 techniques that are considered, they are-

1) Introducing local extensions
2) Replacing type codes with strategy
3) Duplicating the observed data
4) Replacing conditionals with polymorphism
5) Replacing type codes with sub-classes
6) Extracting sub-classes
7) Using form template method
8) Extracting Interfaces
9) Introducing few null objects
10) Using push down method

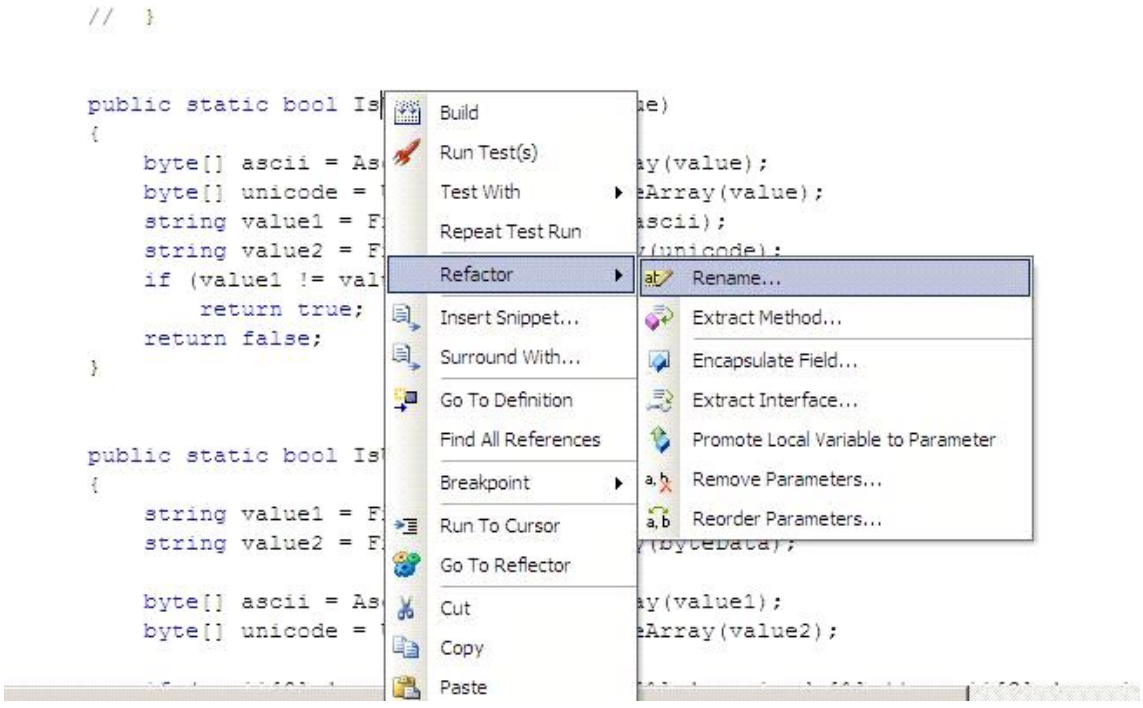


Fig 3: Refactor tool available in a software.

A. Analysis of Data – Internal Measures

The data for empirical study was produced from both non-re-factored and re-factored codes. Following are the methods/formulas considered by the tools that generated the code metric data used for the analysis.

1) *Maintainability Index (MI):* A software metric which measures how source code is maintained.
   MI MAX (0, (171 - 5.2 * ln (Halstead Volume) - 0.23 * (Cyclomatic Complexity) - 16.2 * ln (Lines of Code)) *100 / 171).
2) *Lines of Code (LoC):* Computes the number of lines in a code sample.
3) *Depth of Inheritance:* It determines the extent of inheritance each class possess based on the class hierarchy.
4) *Cyclomatic Complexity:* It computes the complexity of the structures of a code sample by measuring the number of code paths in the program flow.
5) *Class Coupling:* This explains how one class is connected with another class. The extent of coupling determines the code quality. A high class-coupling indicates a low software code quality. The parameters used for measuring coupling are return types, local variables, base classes, parent classes, interfaces used, function declarations etc.

Table 1 Empirical analysis values for all the re-factoring scenarios.

Internal Measures	Non-Re-factored Code	Re-factored Code	% Change in the code quality
Maintainability Index	73	76	4.5%
Lines of Code	4944	5027	-3%
Depth of Inheritance	7	7	0%
Cyclomatic Complexity	1387	1456	-5.3%
Class Coupling	224	240	-6%

From the above table it can be observed that the non-re-factored code had a higher maintainability index than re-factored code. Therefore, it can be concluded that re-factoring enhances the maintainability index of the software code.

However, when maintainability is enhanced, this indirectly decreases the other internal measures. Cyclomatic complexity and class coupling of non-re-factored code sample were less when compared to re-factored code. Lower values of these internal measure are desired. The number of lines of code are increased on re-factoring. Although depth of inheritance is not increased or decreased, other internal measures have seen an overall decrement. The table summarized our empirical study thereby giving a quantitative analysis of the effect of re-factoring on software code quality. It can be concluded that internal measure are not enhanced by re-factoring.

B. Discussion

The above conclusion was drawn from quantitative analysis. The effects of re-factoring on internal measures of software code are taken into consideration. Below is the table that represents the overall findings of empirical study.

Table 2 Tabular view of effect of re-factoring on internal measures

Internal measures	Not enhanced	Enhanced
MI		•
LoC	•	
Depth of Inheritance	•	
Cyclomatic Complexity	•	
Class Coupling	•	

MI is a composite number that depends on various other parameters. One such parameter is the Halstead Volume. Although Cyclomatic complexity is a separate internal measure, MI is dependent on its value. The LoC indirectly has an effect in determining the MI. Comment lines have no effect on the program execution time or complexity. However, they have slight effect on the overall code maintainability. The study has shown an improvement in the maintainability, but it has also shown a decrement in the measures on which it depends. Since MI is just a composite number, it is possible that the value might not be accurate all the time. Hence, drawing conclusions on the basis on MI is not appropriate.

Therefore, it can be concluded that re-factoring does not enhance internal attributes of a software code.

C. External Measures

According to the empirical studies, over all analysis of the code reveals that there is an improvement in the code quality. However, software code quality is a very generic term. There are no standards to determine the quality of a code. Each internal and external measure is validated over each change and finally an approximation is made to check for improvement. In this study, validation is done for five external measures.

1) *Maintainability:* The extent of difficulty in incorporating changes to a section of the code is determined by the maintainability of the software code.

To check the maintainability, few sub characteristics need to be checked

a) Changeability
b) Analysability

2) *Efficiency:* A software quality is tested and validated against many parameters. The major parameters considered are the level of performance and the efficiency in utilising the resources like memory allocated, processors allocated, etc.

To check efficiency, the following sub characteristics are checked

a) Time behaviour
b) Resource Utilization

D. Analysis Of Data – External Measures

1) Analytical study of Changeability

The difficulty in incorporating a change in a section of the code was determined by randomly inserting two changes. The time needed to debug these changes was measured. By the analysis performed it is determined that -

"Changeability of refactored code is much easier than non-refactored code".

2) Analytical Study of Analysability

The outcome for analysability analysis is that -

"Analysability of refactored code is greater than non-refactored code".

3) Analytical Study of Time Behaviour

Time behaviour was measured by recording the task execution time. A piece of code which is highly affected by re-factoring treatment was selected and the task which is related to that code segment was selected for testing. Both pre and post refactored programs were modified to execute 1000 times automatically, and the results were recorded in milliseconds. Outcome of Time Behaviour analysis is that-

"Response time of refactored code is faster than non-refactored code".

4) Analytical study of Resource Utilization

The efficiency of utilising allotted resources is measured by computing the memory utilisation and time behaviour of the code, while it is being executed. A section of code was selected and the task which is related to that code segment was selected for testing. Both pre and post refactored programs were changed to execute 1000 time automatically and the results were recorded in bytes. Outcome of Resource Utilization analysis is that -

"Efficient utilization of System Resources is higher for refactored code than non-refactored code".

IV. DISCUSSIONS

In [1], Källén, M., and Holmgren, suggested that extensive empirical study should be carried out software developed using different object-oriented programming languages, and thus a universal tool should be developed that can point out the possible code elements that can be re-factored and also perform the refactoring operations on the selected code segment. He also stated that a lot of research is currently being done, to explore the effect of re-factoring on software code quality.

In [2], Al Dallal J and Abdin A stated that mostly different re-factoring techniques have the potential to give different outcomes each time they are applied. They might also produce conflicting outcomes. Therefore, the author suggested that comparison of code quality before and after applying refactoring techniques is not productive. He also concluded that re-factoring does not always enhance the code quality.

In [3], Kannangara & Wijayanayake evaluated 10 re-factoring techniques. He conducted various experiments to determine the external measures such as Changeability of the code, Utilization of the resources, Analysing capacity, and Time Behaviour. He also tried to determine few internal factors such as class coupling, cyclic complexity, lines of code (LOC), depth of inheritance and maintainability index. He concluded that there were no significant improvements shown in the external measures, thus there is no impact on the code quality.

In [4], Al Dallal found that the basic context is identifying re-factoring opportunities in object-oriented code prior to the actual re-factoring process. The author drew his conclusions based on analysing the problem statements using various criteria. These criteria includes the re-factoring techniques, processes of identifying the re-factoring possibilities, the selection of usable data sets, and ways of performing the empirical study

V. CONCLUSION

The objective of this empirical study was to explore the effects of re-factoring on software code quality. In order to have a valid conclusion the study was conducted separately on internal and external measures.

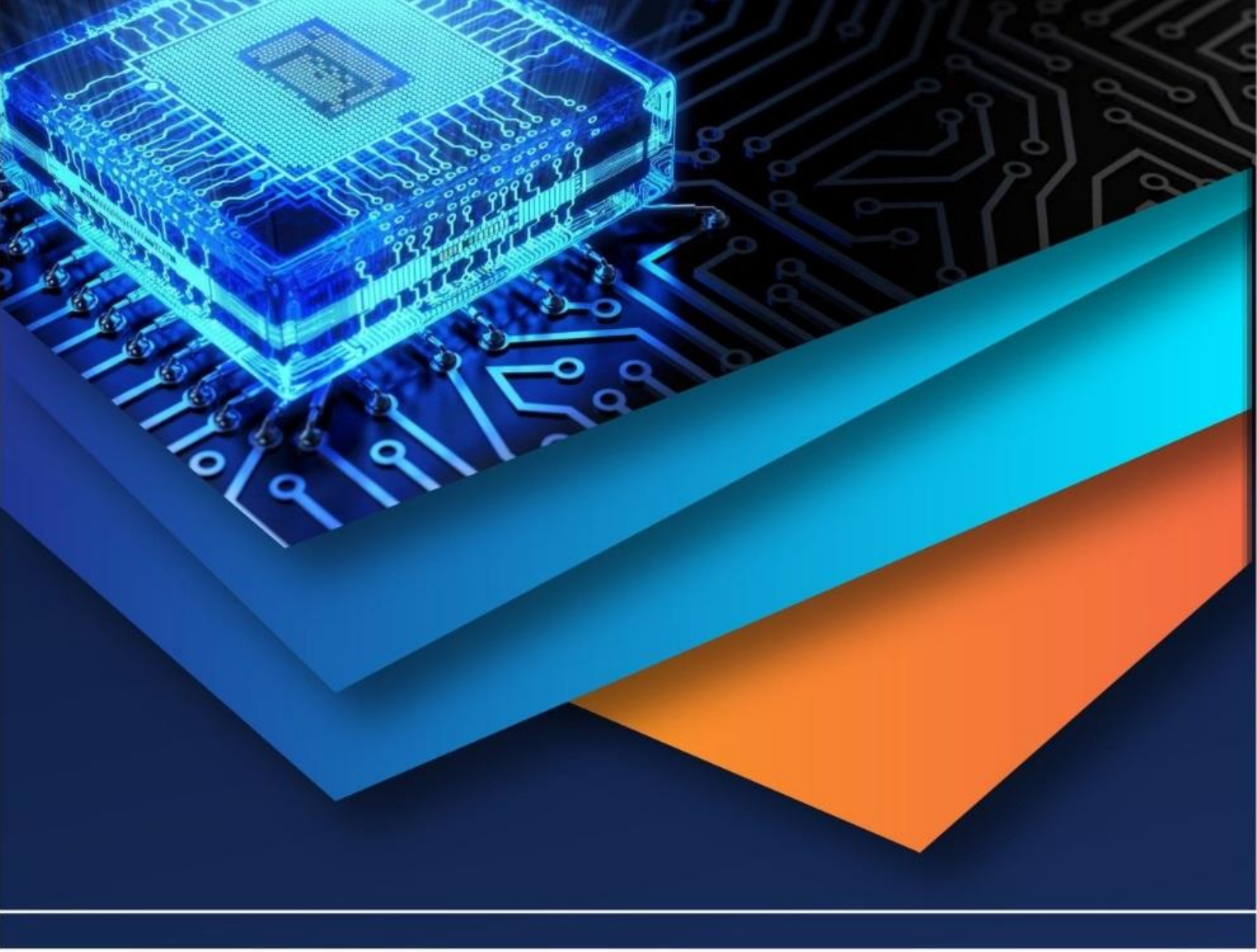
Initially re-factoring techniques were validated against internal measures taking into consideration of a few ISO standard measures like MI, LoC etc. Followed by this, re-factoring techniques were validated against external measures such as changeability, resource utilisation, etc. The empirical study approach was used to perform a quantitative analysis.

The empirical study of the obtained metric values revealed that re-factoring has no significant impact on the external measures. The four considered external measures remained unaffected by the application of any of the 10 re-factoring techniques. When internal measures were considered, although maintainability showed an improvement, the metrics on which it is dependent on showed an overall decrement. Although re-factoring enhanced the given internal measures to a minimum extent, external measures show no improvement, wherein they have a great impact on the software quality.

Thus, we can conclude that according to the results of both empirical studies using two measures namely external and internal measures, re-factoring does not produce any significant improvement in the software code quality

REFERENCES

[1] Källén, M., Holmgren, S., & þóra Hvannberg, E. (2014, September). Impact of code re-factoring using object-oriented methodology on a scientific computing application. In 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation (pp. 125-134). IEEE.
[2] Al Dallal, J., & Abdin, A. (2018). Empirical evaluation of the impact of object-oriented code re-factoring on quality attributes: a systematic literature review. IEEE Transactions on Software Engineering, 44(1), 44-69.
[3] Kannangara, S. H., & Wijayanayake, W. M. J. I. (2015). An empirical evaluation of impact of re-factoring on internal and external measures of code quality. arXiv preprint arXiv:1502.03526.
[4] Al Dallal, J. (2015). Identifying re-factoring opportunities in object-oriented code: A systematic literature review. Information and software Technology, 58, 231-249.
[5] Jabangwe, R., Börstler, J., Šmite, D., & Wohlin, C. (2015). Empirical evidence on the link between object-oriented measures and external quality attributes: a systematic literature review. Empirical Software Engineering, 20(3), 640-693.
[6] Mens, T., & Tourwé, T. (2004). A survey of software re-factoring. IEEE Transactions on software engineering, 30(2), 126-139.
[7] Kannangara, S. H., & Wijayanayake, W. M. J. I. (2015). An empirical evaluation of impact of re-factoring on internal and external measures of code quality. arXiv preprint arXiv:1502.03526.
[8] Alshayeb, M. (2009). Empirical investigation of re-factoring effect on software quality. Information and software technology, 51(9), 1319-1326.
[9] Kumar, R. P. (2017). Internal and External Measures of Code Quality Impact on Re-factoring. SSRG Int. J. Civ. Eng.–ICCREST'17, 70-72.
[10] Pressman, R. S. (2005). Software engineering: a practitioner's approach. Palgrave Macmillan.
[11] Sillitti, A., Janes, A., Succi, G., & Vernazza, T. (2003, September). Collecting, integrating and analyzing software metrics and personal software process data. In null (p. 336). IEEE.
[12] Leopold, H., Smirnov, S., & Mendling, J. (2012). On the re-factoring of activity labels in business process models. Information Systems, 37(5), 443-459.
[13] Crispin, L. (2006). Driving software quality: How test-driven development impacts software quality. Ieee Software, 23(6), 70-71.



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY