IMPROVING THE QUALITY OF SOFTWARE BY REFACTORING

Dissertation submitted in fulfilment of the requirements for the Degree of

MASTER OF TECHNOLOGY

In

COMPUTER SCIENCE AND ENGINEERING

By

GURPREET KAUR

11506686

Supervisor

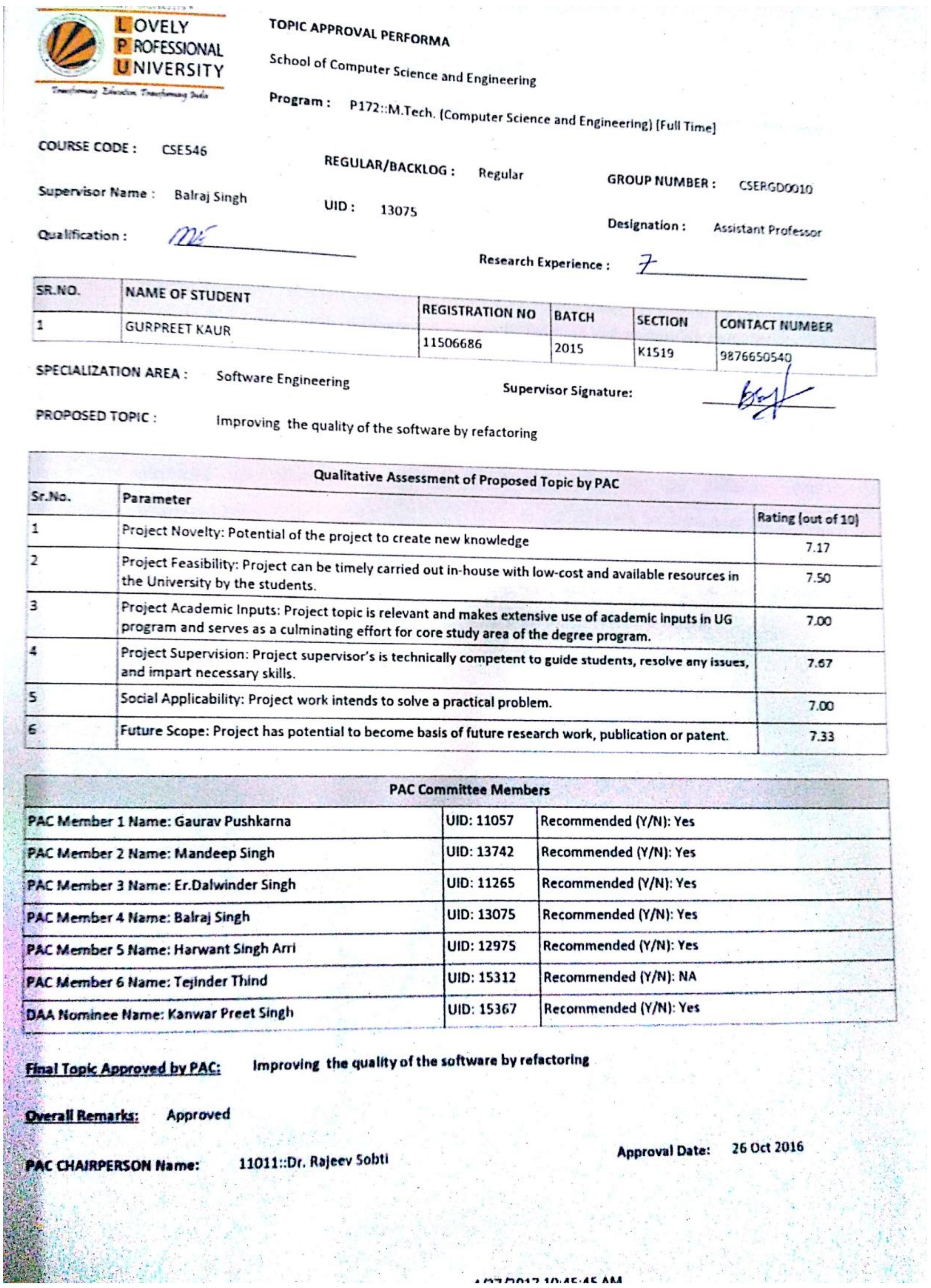
Mr. BALRAJ SINGH

School of Computer Science and Engineering

Lovely Professional University

Phagwara, Punjab (India)

May, 2017



LOVELY PROFESSIONAL UNIVERSITY

Transforming Education, Transforming India

TOPIC APPROVAL PERFORMA

School of Computer Science and Engineering

Program : P172::M.Tech. (Computer Science and Engineering) [Full Time]

COURSE CODE : CSE546	**REGULAR/BACKLOG :** Regular	**GROUP NUMBER :** CSERGD0010	
Supervisor Name : Balraj Singh	**UID :** 13075	**Designation :** Assistant Professor	
Qualification : ME		**Research Experience :** 7	

SR.NO.	NAME OF STUDENT	REGISTRATION NO	BATCH	SECTION	CONTACT NUMBER
1	GURPREET KAUR	11506686	2015	K1519	9876650540

SPECIALIZATION AREA : Software Engineering **Supervisor Signature:** [illegible]

PROPOSED TOPIC : Improving the quality of the software by refactoring

Qualitative Assessment of Proposed Topic by PAC		
Sr.No.	Parameter	Rating (out of 10)
1	Project Novelty: Potential of the project to create new knowledge	7.17
2	Project Feasibility: Project can be timely carried out in-house with low-cost and available resources in the University by the students.	7.50
3	Project Academic Inputs: Project topic is relevant and makes extensive use of academic inputs in UG program and serves as a culminating effort for core study area of the degree program.	7.00
4	Project Supervision: Project supervisor's is technically competent to guide students, resolve any issues, and impart necessary skills.	7.67
5	Social Applicability: Project work intends to solve a practical problem.	7.00
6	Future Scope: Project has potential to become basis of future research work, publication or patent.	7.33

PAC Committee Members		
PAC Member 1 Name: Gaurav Pushkarna	UID: 11057	Recommended (Y/N): Yes
PAC Member 2 Name: Mandeep Singh	UID: 13742	Recommended (Y/N): Yes
PAC Member 3 Name: Er.Dalwinder Singh	UID: 11265	Recommended (Y/N): Yes
PAC Member 4 Name: Balraj Singh	UID: 13075	Recommended (Y/N): Yes
PAC Member 5 Name: Harwant Singh Arri	UID: 12975	Recommended (Y/N): Yes
PAC Member 6 Name: Tejinder Thind	UID: 15312	Recommended (Y/N): NA
DAA Nominee Name: Kanwar Preet Singh	UID: 15367	Recommended (Y/N): Yes

Final Topic Approved by PAC: Improving the quality of the software by refactoring

Overall Remarks: Approved

PAC CHAIRPERSON Name: 11011::Dr. Rajeev Sobti **Approval Date:** 26 Oct 2016

4/27/2017 10:45:45 AM

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 analyze 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.

DECLARATION

I hereby declare that the research work reported in the dissertation entitled " IMPROVING THE QUALITY OF SOFTWARE BY REFACTORING” in partial fulfillment of the requirement for the award of Degree for Master of Technology in Computer Science and Engineering at Lovely Professional University, Phagwara, and Punjab is an authentic work carried out under supervision of my research supervisor Mr. Balraj Singh. I have not submitted this work elsewhere for any degree or diploma.

I understand that the work presented herewith is in direct compliance with Lovely Professional University’s Policy on plagiarism, intellectual property rights, and highest standards of moral and ethical conduct. Therefore, to the best of my knowledge, the content of this dissertation represents authentic and honest research effort conducted, in its entirety, by me. I am fully responsible for the contents of my dissertation work.

Signature of Candidate

Gurpreet Kaur

11506686

SUPERVISOR'S CERTIFICATE

This is to certify that the work reported in the M.Tech Dissertation entitled "**IMPROVING THE QUALITY OF SOFTWARE BY REFACTORING**", submitted by **Gurpreet Kaur** at **Lovely Professional University, Phagwara, India** is a bonafide record of his / her original work carried out under my supervision. This work has not been submitted elsewhere for any other degree.

Signature of Supervisor

Balraj Singh

Date:

Counter Signed by:

1) **Concerned HOD:**
   HoD's Signature: ________________

   HoD Name: ____________________

   Date: ___________________

2) **Neutral Examiners:**

   External Examiner

   Signature: _______________

   Name: __________________

   Affiliation: ______________

   Date: ___________________

   Internal Examiner

   Signature: _______________

   Name: __________________

   Date: ___________________

ACKNOWLDGEMENT

It is not until you undertake research like this one that you realize how massive the effort it really is, or how much you must rely upon the selfless efforts and goodwill of others. I want to thank them all from the core of my heart.

I owe special words of thanks to my supervisor Mr. Balraj Singh for his vision, thoughtful counseling and encouragement for this research on “**IMPROVING THE QUALITY OF SOFTWARE BY REFACTORING**”. I am also thankful to the teachers of the department for giving me the best knowledge guidance throughout the study of this research.

And last but not the least, I find no words to acknowledge the financial assistance & moral support rendered by my parents and moral support given by my friends in making the effort a success. All this has become reality because of their blessings and above all by the grace of almighty.

Gurpreet Kaur

TABLE OF CONTENTS

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Checklist for Dissertation-III Supervisor

Name: ____________________________ UID: ________ Domain: _____

Registration No: ____________Name of student:___________________

Title of Dissertation: ____

- ☐ Front pages are as per the format.
- ☐ Topic on the PAC form and title page are same.
- ☐ Front page numbers are in roman and for report; it is like 1, 2, 3…….
- ☐ TOC, List of Figures, etc. are matching with the actual page numbers in the report.
- ☐ Font, Font Size, Margins, line Spacing, Alignment, etc. are as per the guidelines.
- ☐ Color prints are used for images and implementation snapshots.
- ☐ Captions and citations are provided for all the figures, tables etc. and are numbered and center aligned.
- ☐ All the equations used in the report are numbered.
- ☐ Citations are provided for all the references.
- ☐ **Objectives are clearly defined.**
- ☐ Minimum total number of pages of report is 50.
- ☐ Minimum references in report are 30.

Here by, I declare that I had verified the above mentioned points in the final dissertation report.

Signature of Supervisor with UID

CHAPTER 1

INTRODUCTION

Software is used to manage to solve the real world problems and so that it makes changes in generated problems. Software is developed using different phases of software life cycle models. Different models are used to develop software's. For example waterfall model, iterative model, spiral model, evolutionary model, prototype model, incremental model and agile model etc. software development include different phases like requirement gathering , design and code , implementation , testing and maintenance phase. At implementation and maintenance phase we can change and improve the code. To change and improve the code we add and remove some features and requirements.

1.1 REFACTORING

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.

1.2 REFACTORING PROCESS

The refactoring process contains three major aspects: identification of refactoring candidates, validation of refactoring effect, and application of refactoring. The three important roles in the refactoring process are: the developer, the analyst, and the manager. To begin the process of refactoring the code would be analyzed by the developer to check which part to be refactored. Those refactoring candidates are to be examined by the analyst in terms of the cost and effect [5]. After the developer identified the refactoring process, analyst selects the relevant refactoring method to refactor the source code. And project manager settle on choices producing the cost results.

1.2.1 Planning

i. "Bad–smell" is characterized as a program trademark which demonstrates to the need of program refactoring. For instance, "copied code" is one sort of bad–smell in light of the fact that there is an opportunity to improve the code by join those copied parts [15].
ii. "Bad–smell" Analysis "Bad–smell" does not really prompt to an individual specific refactoring. As a rule, copied codes are to be bound together. It is ideal; in any case, that such a brought together code is actualized in a super class instead of other subjective spots if the duplication is just found among sibling sub classes [15]. By breaking down bad–smells decisively, we can distinguish a superior arrangement.
iii. Refactoring Planning: After examining different bad–smells, various refactoring candidate would be distinguished. Some will be simple to perform and some will be

difficult to figure it out [5]. Besides single applicant could be a partner of another so those two refactoring can't applied at parallel time. For example, "Remove Method" turns a code part which can be gathered together into a technique whose name clarifies the reason for the strategy [5].

1.2.2 Validation

i. Plan Evaluation: A refactoring course of action ought to be surveyed as cost and effect. As a rule, a software project can manage the cost of just a restricted measure of asset to perform refactoring Meeting the delivery due date with involved requirements is considered as the primary goal.
ii. Refactoring Validation: Bad–smell analysis produces a set of refactoring candidates. Refactoring planning then sift out preferable refactoring candidates from the entire set. It is reasonable to validate those sifted candidates in terms of Detected bad–smells again to confirm.
iii. Functional Equivalence Validation: Essentially refactoring ought not to affect the usefulness of the objective program. In this way functional equality approval may be required after all the refactoring have been connected.

1.2.3 Execution

i. Refactoring Deployment: The refactoring arrangement has been developed; it must be sent as program adjustments. The specialist is capable to this stage: He/she needs to "shape" the refactoring undertaking.
ii. Refactoring Application: Every designer is able to the assigned refactoring. Frequently designs need to examine their own particular assignment to each other to stay away from inverse program changes.

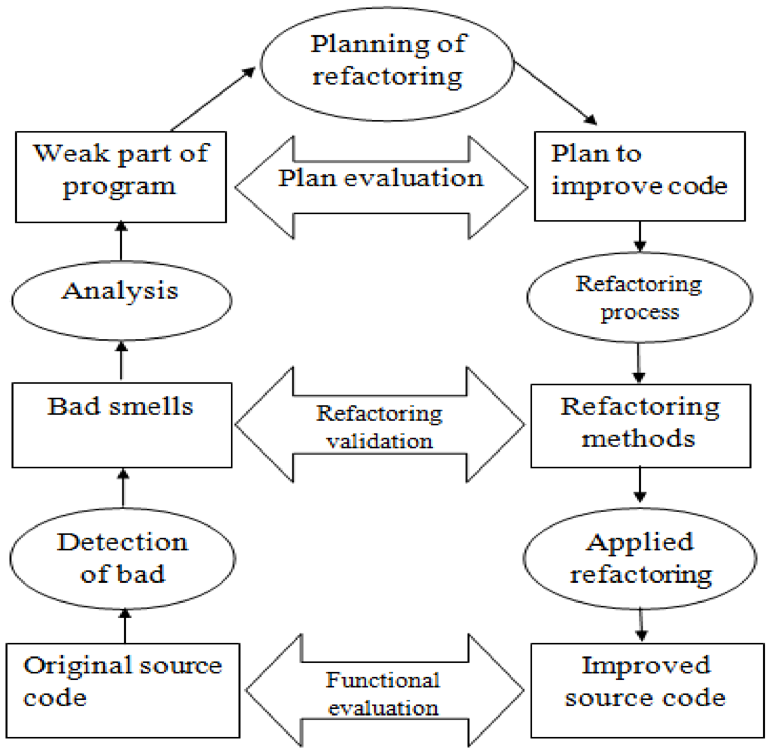


Figure1.1. Refactoring process

1.3 BAD SMELLS: The bad smells in the code are defined as follows:

Table 1.1: signs of code that might need refactoring

Name	Description	Solutions
Duplicate Code	At the point when same code components exist in various places as opposed to one place, it is a copy code. Duplicate code is quite common. Duplicate code become incorrect because if you improve or change individual instance of copied code but not the others, you may have introduced a flaw. E.g. having a similar expression in two related	Extract method, Extract class, pull up method

	subclasses.	
Long Method	A method contains too many lines of code.	Extract method, Replace temp with query
Long class	Understanding and keeping up classes dependably costs time. So if a class doesn't do what's necessary to win your consideration, it should be deleted.	Extract Class, Extract Subclass, Extract Interface, Replace Data Value with Object
Long Parameter list	More than three or four parameters for a method and make code harder to understand.	Replace parameter with method, Introduce parameter object
Feature Envy	At the point when another class is depending upon another to give a particular functionality, another class may need to complete that functionality.	Move method, Extract method, Move field
Switch Statements	Number of switch statements shows procedural statements.	Replace condition with Polymorphism, Replace Code with Subclasses, , Replace Parameter with Explicit methods, Introduce null objects

1.4 REASONS OF REFACTORING

i. Reuse mechanism

One can reuse requirement, code, design, and test case in any period of the software development cycle.

ii. To meet the deadline

Time constraint leads to the software or code refactoring. Most of the programmers in companies do copy and paste or make certain amendments in code in order to meet deadline and to achieve desired functionality.

iii. Lack of Interpretation of requirements

It is hard to translate and make an orderly approach for every single prerequisite as a outcome of the number of determinations in extensive frameworks.

iv. Tested code

As there is always risk associated with new code because programmer can develop the code which might be more mind boggling or more inclined to bugs and errors. So to copy code is always preferable choice.

v. Less knowledge of the new language

Sometimes programmer does not have the better command over the programming language.

1.5 ADVANTAGES OF REFACTORING

i. Improves the design of software

Refactoring is used for improving the code is directly measured the software metrics and indirectly measured the software quality attributes.

ii. Makes software easier to understand

Refactoring makes software more readable form. It helps to understand unfamiliar code.

iii. Minimize code duplication

Modify the inner structure and make it simple to understand.

iv. From messy to clarity

Where your code might start off as messy code, before you start refactoring, the Clarification process should slowly untangle that mess into clear steps that reflect what is going on in clear steps.

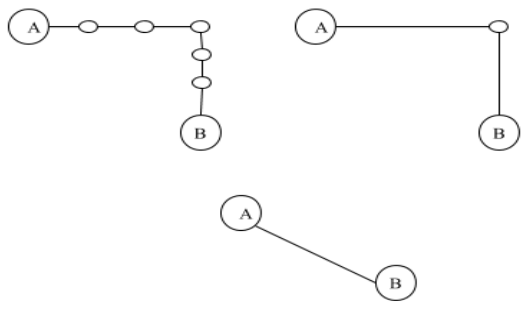


Figure1.2. An *example: counting and reducing the steps from request (A) to concrete execution (B)*

1.6 DISADVANTAGES OF REFACTORING

i. Refactoring on code support step-by-step activities over a relatively extra period of time.
ii. It takes time and if it done incorrect, it produces unwanted tight couplings between discrete modules of the system and makes things even more complex. If you don't create a healthy set of tests to back you up then you can break things too.

1.7 TECHNIQUES OF REFACTORING

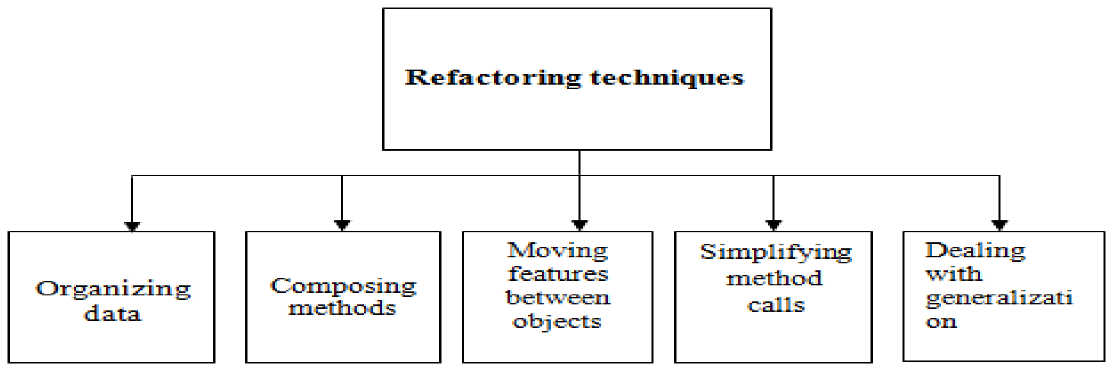


Figure1.3. Refactoring techniques

1.7.1 Organizing data

In this technique replace the primitives of the class with the rich class which makes classes more reusable and portable. Following are the methods to organize data [14]:

i. Replace methods with method objects: Remove a procedure from a set of selected declaration to a new class; contain the removed statement as a new procedure and local variable of the procedure as field of new class

ii. Replace constructor with factory method*:* Hide constructor and exchange it with static technique which returns new instance of the class.

1.7.2 Composing methods

In this technique the code inside these methods hide the method of execution and make the code extremely hard to understand and change. Following are the methods to organize data:

i. Extract method: Extract a code into small pieces to create new methods. Extraction of a bit of code into a different strategy. You have a code section that shows up in numerous spots inside the code .You have a code part that can be assembled together. Transform this part into a technique whose name clarifies the reason for this strategy.

ii. Split temp variable: Temporary variables is allocate to more than once and not a loop variable. Sometimes a local variable can take on various identities, assuming distinctive parts through the duration of a strategy. This makes code harder to understand and read, on the grounds that the part of that variable relies on upon its position in the code. This refactoring cleans up the chaos (mess), making another local variable for the task.

1.7.3 Moving features between objects

This refactoring technique shows how to move functionality of the classes safely, create new classes and hide the details of implementation from the public access. Following are the methods used to move features between objects:

i. Extract class: Generate new classes and moves to it new method from obtained classes. The Extract Class refactoring is connected when a class gets to be distinctly overweight with an excessive number of strategies and its motivation gets to be distinctly unclear. Extract Class refactoring includes making another class and moving techniques as well as information to the new class.

ii. Move method: these refactoring permits moving a technique starting with one class then onto the next. The need of transfer a procedure comes when the strategy is utilized in different class than the class in which it is characterized.

1.7.4 Simplifying method calls

It makes method call simpler and easier to understand. It simplifies the interface for interactions between classes.

i. Delete class: Remove the class and its references.
ii. Delete attribute: Remove an attribute that is not referred to any class.
iii. Delete method: Simply remove the procedure which is not belongs to any class.
iv. Rename method: Modify the name of a method and modify it from all the locations where it is referred.

1.7.5 Dealing with generalization

Fundamentally connected with moving usefulness along the class legacy progressive system, making new classes and interfaces.

i. Pull up attribute: Transfer an attribute to super class or parent class of the present class.
ii. Push down method: Transfer a procedure to one or more sub class. The Push Members down refactoring permits in moving the class individuals into subclass/sub interface for cleaning the class hierarchy.
iii. Extract Subclass: A class has attributes that are utilized just in a few cases. Make a subclass for that subset of elements. For example class A has fields that are used some instance only. To solve this problem we simply create a class B, subclass of A. class B will be used in the case an instance would need all fields from original class A, new class A will be used otherwise.
iv. Extract Superclass: Having different classes with similar elements. Make a root class what's more; transfer the regular components to the parent class.

1.8 SOFTWARE QUALITY ATTRIBUTES

Software quality is the standard to which software hold a required combination of attributes (e.g., reliability, interoperability). Software quality attributes include scalability, security, performance and reliability. Quality attribute requirements are unit of an application's nonfunctional requirements, which represent the many features of how the functional requirements of an application are execute. The quality term defines different meanings. Basically it depends upon the user, customer and developer of the system. The goal of the developer collect those requirements and which satisfy the customer needs. Quality attributes are register for both the product and the process. Product defined as which delivered to the customer and process defines as which manufacture the software product.

1.9 SOFTWARE METRICS

Software metrics are also used as the internal quality attributes. Software metric is better occurrence of measuring the quality of software. Software metrics provide a mode to extract useful and quantifiable material about the construction of the software. The software program has a list of metrics in order to assume the structure and the quality of the system. Measuring the complexity of the system is the common procedure to estimate the maintainability of the software. If the estimated result has the higher value the program is complex and not easy to maintain.

Following are the quality metrics that we are going to use in our research:

Weighted methods per class (WMC), it is used to describe the number of methods that are used in specific class. Normally, it is used to calculate the complexity of an individual class. Response set for classes (RFC), Number of methods that can be complete in response to a message being received by an object of that class. Number of outgoing invocations (NOI), total comment line of code (TCLOC), Total logical lines of code (TLLOC), and total number of statements (TNOS), clone instance (CI), metrics are used in our research. Maintainability metrics that we are used: maintainability index, cyclomatic complexity, depth of inheritance and line of code.

The metrics we investigate are the following:

i. **Depth of Inheritance Tree (DIT)**: It defines the length of the extended path from a node to the parent class in the inheritance hierarchy. Main purpose of DIT is decomposition [1].

ii. **Number of Children (NOC)**: defined as the no. of classes that inherit directly from a given class [9].

iii. **Response for a Class (RFC)**: defined as the number of procedures that can be executed in response to a message accepted by an object of that class [1]. RFC is the number of local methods plus the number of procedures called by local methods.

$RFC = |RS|$

Where, RS is the response set for the class

"Response set of an object ≡ {set of all methods that can be invoked in response to a message to the object}"

iv. **Weighted Methods per Class (WMC)**: WMC is the sum of the complexities of the methods; complexity is measured by cyclomatic complexity [1].

$WMC = \sum^{n}_{i=1\ Ci}$

Where, a class Ci has M_1....M_n, methods with C_i.....$C_{n,}$ complexity respectively.

v. **Number of Methods (NOM)**: defined as the number of procedures perform or execute in a specific class [14].

vi. **Lines of Code (LOC)**: defined as the total no. of Lines of source code in a class and exclude all empty and Comment lines [17].

CHAPTER 2

REVIEW OF LITERATURE

Software or code refactoring has become a major area of research these days. Many researchers diligently exploring this topic and so many approaches have been developed to probe duplicate codes.

Study from 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 connection between refactoring and quality of the code. The research has organized three java system software with 63 releases 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]. To measure the effect of refactoring they used coupling metrics and selected 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. The author used some refactoring techniques like composing method which includes extract method, replace temp with query, inline method refactoring methods, moving feature between object includes move method, inline class, for organizing code uses replace code with class, change value to references, and replace array with object for 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]. Basically in this paper they used refactoring methods to improve the maintainability and software implementation of targeted software in order to choose a safe process. For implementing this method they used refactoring tool name as Refactoring Assistant.

Study from Michael Wahler *et.al* in 2016[33]. In this paper the author defines a case study in which magnetic researchers were discussed by software engineers in refactoring. The shareholder of the research product considered the software to be un-maintainable as it had reached to a size of 30 kilo line of code of Java. The study states that the procedure of refactoring the product under the advice of a software engineer with supported results by static analysis and software metrics. They propose a case study on refactoring a design tool for increasing the maintainability of code using magnetic components. In order to prioritize the maintenance tasks, they combined the results from automatic code analyses with the individual assessment of the original developer. The number of future obstacles found by Find Bugs was minimized by 23 % and around 82% of amount of replicated lines of code was minimized.

Study from Chaitanya Kulkarni *et.al* in 2016[34]. In this paper the author aims mainly towards the chances of detecting a refactoring code and to find out whether the code clone can be assured refactored or not. Three methods were tried: 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 different procedures and also removes the problem of code cloning through refactored the code.

A comprehensive study of the different techniques of refactoring was also done which made it easier for the programmers to get to know the code. Outcomes showed that refactoring of code can remove the limitations which occur due to code clone.

Study from Minas F. Zibran *et.al* in 2015[35]. In this paper author tells about characteristics of clones can be understood by clone analysis and visualization. They indicate potential clones as cost-effective candidates for refactoring. 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.

Around 80% of the software costs are spent on maintenance. During a maintenance task, maximum of the developer's effort is invested in understanding the underlying program structure and source code, while 62% of such effort is typically wasted in investigating irrelevant parts (e.g., source files) of the program. With proper analogies, significant support for clone investigation and representation with respect to the inheritance hierarchy and call charts can help in settling on better plan choices amid clone refactoring and subsequently can limit clone refactoring cost, which in turn can lessen the product maintenance cost all in all.

Study from Anna Vasileva et.al in 2016[36]. They showed the effective combination of calculation of code quality into a software development process. Concepts for removal of inadequacy are significant pre-requisites for code quality besides selecting an appropriate tool for code analysis. In this paper, they showed that implementation of measurement and didactic procedures in several iteration cycles can ensure the long term integration of quality aspects. Simple refactoring techniques are used for example rename were used successfully by all teams. Their investigation showed that the deadline of work with tough refactoring techniques is very complex for developers that are inexperienced. They concluded that good internal quality of program code can be achieved without high efforts or achievements. Quality of the code can be achieved in the starting of the project as early as in the designing phase if the aim is set right. Therefore, the authors focus on the beginning phase in their future research. They further planned to include calculations of model quality aspects and the successful didactic methods in order to enhance the modeling outcomes.

Study from S.H. Kannangara *et.al* in 2013[37]. The goal of this paper was to prove the request that refactoring increases quality of the software. The objective was achieved by utilizing the experimental research approach and for the analysis; selected refactoring techniques were used. The effect of each refactoring was judged based on external estimations, which were; analyzability, time behavior and resource deployment. After analyzing the results of the experiment, “Replace Conditional with Polymorphism” ranked in the highest among the tested 10 refactoring techniques as it showed a high percentage of improvement in code quality. Whereas “Introduce Null Object” was categorized as worst as it deteriorated the code quality in a huge amount. The hypothesis testing results indicated that the analyzability of refactored code is less than non-refactored code for all the tested refactoring techniques except for “Replace conditional with polymorphism”.

From the analysis of four external estimations “Replace Conditional with Polymorphism” was categorized highest as it had a high percentage of enhancements in code quality. “Introduce Null Object” was ranked as worst as it is had the highest percentage of deteriorated code quality.

The review to discover impact of refactoring on the product quality properties has a wide extension. Fowler has given 70 sorts of refactoring techniques and each refactoring strategy can be connected to the different programming qualities property. Following are the quality attributes used in the study:

i. **Maintainability**: It is characterized as the modifications with which change is made on set of attributes. The change in the properties may contain from prerequisite to plan. It might be about revision, preventive action and adaptation.
   Formula to calculate maintenance:
   M = (time spent to fix a bug/total development time)*100
ii. **Reusability**: It is defined as the reusable pieces of the software in the other elements or in other software system with small adaptation.
iii. **Testability**: It is characterized as how much programming underpins or supports testing process. High testability requires less exertion for testing.
   T = (time spent to testing the functionality/development time)*100
iv. **Understandability**: It is characterized as the simplicity of understanding the significance of programming parts to the client.
v. **Fault proneness**: Fault Proneness in the projects is more prone to the bugs and breaking down of the module.

vi. **Completeness**: Completeness of the program refers for all the required components, resources, programs and all the possible ways for execution of the program.
Completeness = (no. of requirement full filled/ total no. of requirements)*100

vii. **Stability**: Stability represents the capability of the program to bear the risk of all the unexpected modification or alterations.

viii. **Complexity**: In an intuitive framework it is characterized as the trouble of performing different undertaking like Coding, troubleshooting, actualizing and testing the product.

ix. **Adaptability**: Adaptability of the software is taken in terms of its ability to consume the changes in the system without any arbitration from any external resource.
Sa = ((Rp - Rt)/R1)*100 where,
Sa: software adaptability
Rp: code executed successfully
R1: part of code fails to execute
Rt: total lines of code

Study from S.H. Kannangara *et.al* in 2013 [38] .In this research paper the author presented a way for unifying and refactor the software clones that controls the short comes of earlier approaches. More precisely, their technique was to be able to find and limitations of the non-trivial differences among the clones. Moreover, it detects an optimal mapping among the assertions of the clones that reduces the extent of dissimilarity. They differentiate the given technique with a moderate clone refactoring tool and concluded that their perspective was able to detect a notably greater no. of clones that were refactorable.

Study from Tom Mens *et.al* in 2004[39].This research paper gives an overview of existing researches in the domain of software restructuring and software refactoring. The authors organized this research according to different criteria: the supported refactoring activities, the formalisms and specific techniques that are used to support these activities, the types of software artifacts that are being refactored, significant characteristics that needs to be taken into account when refactoring tools are to be build, and the effects of refactoring on the process of software development. In all of the respective categories, they pointed out the necessary open issues that are still to be solved. In general, they found a need for processes, formalisms, tools and methods that address refactoring in a more flexible, consistent, scalable, generic and flexible way. Although proliferation of commercial refactoring tools has begun, research into software refactoring and restructuring continues to be very active and remains essential to finding and solving the limitations of these tools.

Study from Diego Cedrim *et.al* in 2016[40].First longitudinal study was reported in intention to address this variance. They decompose how often the frequently-used refactoring types influence the density of 5 types of bad smells along the version of 25 projects. Their discoveries are rooted upon the review upon 2,635 refactoring distributed in 11 different types. Total count of 2,506 refactoring (95.1%) did not reduce or introduce code smells. As a result, it came out that refactoring lead to smell reduction less often than what has been reported previously. Data conveys that only 2.24% of refactoring changes removed code smells and 2.66% introduced new ones. Several smells were induced by refactoring that tended to live long, i.e., 146 days on average. When smelly elements started to exhibit poor structural quality and, as a consequence, started to be more costly to get rid of, these smells were only eventually removed. We also presented new findings not reported in their previous study [4]: (i) the negative refactoring occur as frequent as the positive ones; (ii) code smells tend to live long (146 days, on average); and (iii) while the software evolves, the existing smells tend to become more complex, increasing the effort of removal.

Think about from Debarshi Chatterji et.al in 2013[41].An broadened replication of a controlled analysis (i.e. a strict replication with an extra tasks) that breaks down the impacts of cloned bugs (i.e. bugs in cloned code) on the program comprehensive of software engineers has been represented. The review members endeavored to disengage and settle two sorts of bugs, cloned and non-cloned, in one of two little frameworks were separated and settled by the members in the strict replication divide. Members are given a clone report depicting the area of all cloned code in the other framework and asked them to again detach and settle cloned and non-cloned bugs in the augmentation of unique review. The cloned bugs were not fundamentally more hard to keep up than non-cloned bugs turned out therefore of the first review. On the other hand, the consequences of the replication demonstrated that it was essentially harder to accurately settle a cloned bug than a non-cloned bug. Be that as it may, there was no critical contrast in the measure of time required to settle a cloned bug versus a non-cloned bug. At last, the consequences of the review expansion demonstrated that developers performed fundamentally preferable when given clone data over without clone data.

The extended replication consisted of two parts: (1) a sound replication of the last research and (2) an addition to the previous study. The aim was to give a perception of the questions present in the actual study by verifying its solutions by giving extra outcomes that could help to understand the impact of the clones on maintenance of the software. The end results of the authentic study showed some trends, but most of the results weren't significant. The results of the replicated study were also not able to verify (1).The results of the replicated study did shoed that it was comparatively tougher to maintain cloned bugs completely. In the Extended portion, the authors found that when creators were provided with clone statics and trained how to use it, helped for maintenaning the bugs. They establish that the participating developers establish better in fixing the bugs completely when they had been provided with clone instructions than when they didn't have the clone information. At last, the evidence from the extended replicated study indicated that it was tougher to maintain cloned code compared to a non-cloned code. However, by providing appropriate information about the clone along with the proper training can reduce this difficulty.

Study from Mesfin Abebe *et.al* in 2014[42].The main motive of this research study is to enlarge a previous research by considering more literatures and using a well ordered method to inspection of literature to improve the validity and accuracy of the research. The authors studied a number of literatures from various databases which were publicizing since 1999 in order to conclude and evolve the knowledge about software re-engineering. The research pattern can be revealed by classification and summarization. The general involvements and statistics of the published papers in the last years can also be considered. The researcher's time and effort can be saved by formulating better research topics with the help of the extracted information. Those research papers then can be used to solve some crucial problems.

From the past fifteen years researcher's have contributed an extensively to the topic of software refactoring, but still there are a huge deal of obstacles that an unresolved till date, which needs to be solved in the upcoming researches. Therefore, the detected gaps and the important subscription can help the researcher's by guiding them where their focus should be. This can help the researchers save effort, time and services. Lastly, this survey can be continued in the upcoming researches by using documented data apart from the literature available in the electronic databases. Moreover, this research can be integrated with practically in the industry of the technology and can be used in future to increase the credibility and maintainability.

Study from Ladan Tahvildari *et.al* in 2004[43].A re-constructing process model and a modified framework was presented by this paper. Determination of delicate objective prerequisites for the objective transient framework and a rundown of programming changes that positively affect such necessities were basically engaged. The recognizable proof of blunder inclined code utilizing measurements and the determination of the fitting changes that can possibly improve the objective qualities and prerequisites for the new framework were focused upon. This paper proposes a structure for consequently recognizing circumstances for specific changes to be connected with a specific end goal to enhance particular outline quality attributes, diverse protest arranged measurements can be utilized as markers. In view of both on displaying the conditions between outline qualities and source code highlights, and on analyzing the effect that different changes have on programming measurements that evaluate the plan qualities being made strides. To anticipate loss of practicality amid advancement by and large or reestablish it through reengineering, this methodology can be utilized.

To examine the utilization of measurements with setting and space particular data can be coordinated to do in future. Refining the determination of suitable changes by killing those that are not important or don't contribute towards the chose qualities being improved.

Study from Eduardo Fernandes *et.al* in 2016[44].This review study depends on three metrics - recall, agreement and precision and two software systems for comparison. The author's results show that tools provide superfluous detection results for same bad smell. it established qualitative and quantitative data, the authors discussed appropriate obstacle related to usability and proposed instructions for creators to detect tools. Considering 84 tools, they observed that the amount of plug-in and standalone tools are almost equal. In addition to this, the observations tell that Java, C++, and are the top-three most layered programming languages used to detecting bad smells. Greater amount of tools implemented in Java relied on the technique of metric-based identification. Lastly, the survey paper showed that Large Class, delicacy of Code and Long Method are the topmost smells. The important donatives of this paper are as follows. The author's presented a literature review of tools that detect bad code. They found 84 dissimilar tools, and they organized them according to similar attributes, such as detection techniques, detected bad smells. Programming language for detection of smells. They also conducted a review of different tools to find the mostly occurred bad smells that the goal of tool to detect the duplicate code. The comparative study and documented survey, they discussed qualitative data (lessons learned) and quantitative (recall, agreement and precision about the tools.

Table 1.2: Summary of refactoring techniques

Authors	Case Study	Internal Measures	External Measures	Refactoring
Kataoka et al.[26]	C++ program	Coupling	Maintainability	Extract Method and Extract Class
Stroulia and Kapoor et al. [24]	Academic	Size and coupling	Design extendibility	Extract Super class, Extract abstract class

Leitch and Stroulia et al. [29]	Academic and commercial	Code size, number of procedures	Maintenance effort and costs	Extract Method, and Move Method
Tahvildari and Kontogiannis et al.[28]	Four open-source applications	coupling, cohesion, inheritance and complexity	Maintainability	Code Transformations
Bois et al.[27]	open source software	cohesion and coupling	-	Extract Method, Move Method, Extract class Replace Method Object, Replace Data Value Object
Moser et al. [32]	A project in industrial environment	Line of code, Chidamber and Kemerer measures, Effort (hour)	Productivity (LOC)	

Alshayeb et al.[31]	Three Open-source projects	Chidamber and kemerer measures, LOC, FOUT	adaptability, maintainability, understandability, reusability, and testability	Extract Class, Extract subclass, Move class, Extract method Encapsulate Field, Replace Temp with Query
Sahraoui et al.[23]	A C++ program	Inheritance and coupling measures	Fault-proneness	Extract Super class, Extract Subclasses, Extract Aggregate Classes
Tahvildari et al.[29]	A project in open source and industrial environment; both written in C.	Halstead efforts, Line of code, and number of Comment lines per module	Maintainability and performance	Design patterns
Yoshio Kataoka[15]	Enhancement effect of program refactoring	Coupling, Cohesion, Size and Complexity	Maintainability enhancement	Move Method, Replace Temp with Query Extract and Inline Method

CHAPTER 3

PRESENT WORK

3.1 PROBLEM DEFINITION:

Maintainability of software code decreases with increase in features and increase in complexity of code. As more and more features and conditions are added to the methods code becomes more error prone. It is very difficult to keep the source code easily maintainable due to non generic nature of the software products. There is a need to understand and define practices that can help real world problem of code maintainability.

3.2 OBJECTIVE OF THE STUDY:

i. To prepare a code maintainability index using ref-finder's proposed refactoring enhancements and upgradations.
ii. To propose an improved refactoring technique.
iii. To evaluate the proposed code re-factoring technique using Junit open source system and generate new code maintainability index.
iv. To evaluate the effectiveness of new proposed technique with existing technique by comparing them.

3.3 RESEARCH METHODOLOGY

Refactoring is a procedure 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 evaluating the software maintainability. In our research we evaluated project for code refactoring and maintainability of code taking four releases for the project. Ref-Finder tool is used to extract code refactoring differences between releases of project

Following are the steps of proposed methodology:

i. Gather source code from previous dataset.
ii. Scan each release individually for code metrics

iii. Measure code metrics
iv. Apply the enhanced re-factoring techniques
v. Measure code metrics again
vi. Compare result with existing techniques

Here is flowchart depicting methodology to be followed for research:

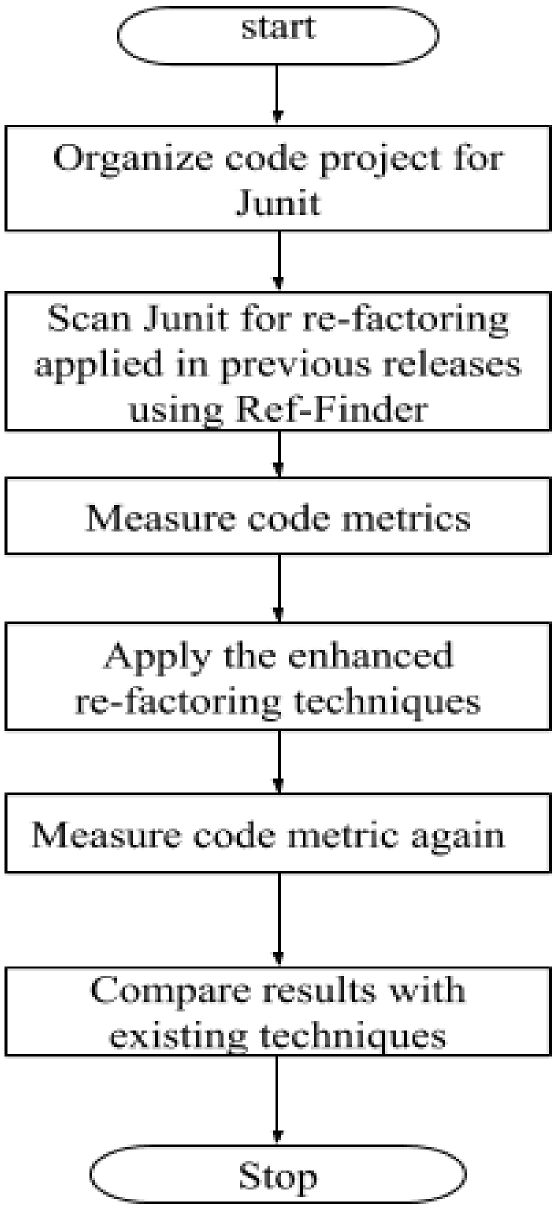

start
Organize code project for Junit
Scan Junit for re-factoring applied in previous releases using Ref-Finder
Measure code metrics
Apply the enhanced re-factoring techniques
Measure code metric again
Compare results with existing techniques
Stop

Figure 3.1: Proposed Methodology

3.3.1 ALGORITHM:

Step 1.Scan Junit releases 4.10, 4.11, 4.12 and 5.0

Step 2. Find following in code scan:

a) Effect of refactoring on low maintenance code?
b) Which attributes of the code are affected most by refactoring?
c) Impact of code re-factoring on future releases like ease of adding new features and removing a feature with minimal changes

Step 3. Create list of refactoring candidate classes

Step 4. For each candidate

d) 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
        1. Flag class as refactoring
    vi. Scan methods for variables used
e) Assign class score for refactored code in variables and methods

Step 5. For each class having score >8 generate list for suggestion of lists for missing refactoring.

3.3.3.1HOW IT WORKS:

a) Initialize ClassName = ClassName, IsRefactoringCandidate = FALSE, RefactoringType= LIST, LineNumber = LIST, MethodFlow, RefactorScore = 0
b) Scan class and list all code fragements
    i. For each class answer 3 questions.
    ii. Generate flow of code:
        1. For each statement scanned, divide statement as:
            a) Assignment: Independent variable assignment

```
            b)  Decision: If-else /switch/ternary operation
            c)  method call: call to method of class
            d)  Loop start: loop
            e)  Prepare flow of class
        2.  If variable assignment does not involve getter/setter methods then
            a)  IsRefactoringCandidate = TRUE
            b)  RefactorScore  = RefactorScore +0.5
        3.  If flow contains more than 5 cases if-else-if OR switch contains 3 cases OR
            Ternary operation has  ladder
            a)  SET IsRefactoringCandidate = TRUE
            b)  Add Inheritance for decisions to Refactoring Type
            c)  Add Line Number to list
            d)  RefactorScore  = RefactorScore +1
        4.  Save method flow in List
c)  For each method:
    i.  Compare flow list with other methods
    ii. If method flow have more than 10 statements common then
        1.  SET IsRefactoringCandidate = TRUE
        2.  Add method to refactoring list
        3.  RefactorScore  = RefactorScore +3
```

CHAPTER 4

RESULTS AND DISCUSSIONS

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.

4.1 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 1.3 gives the aggregate number of classes, methods and refactoring procedures.

Table 1.3: Total number of classes, methods and refactoring

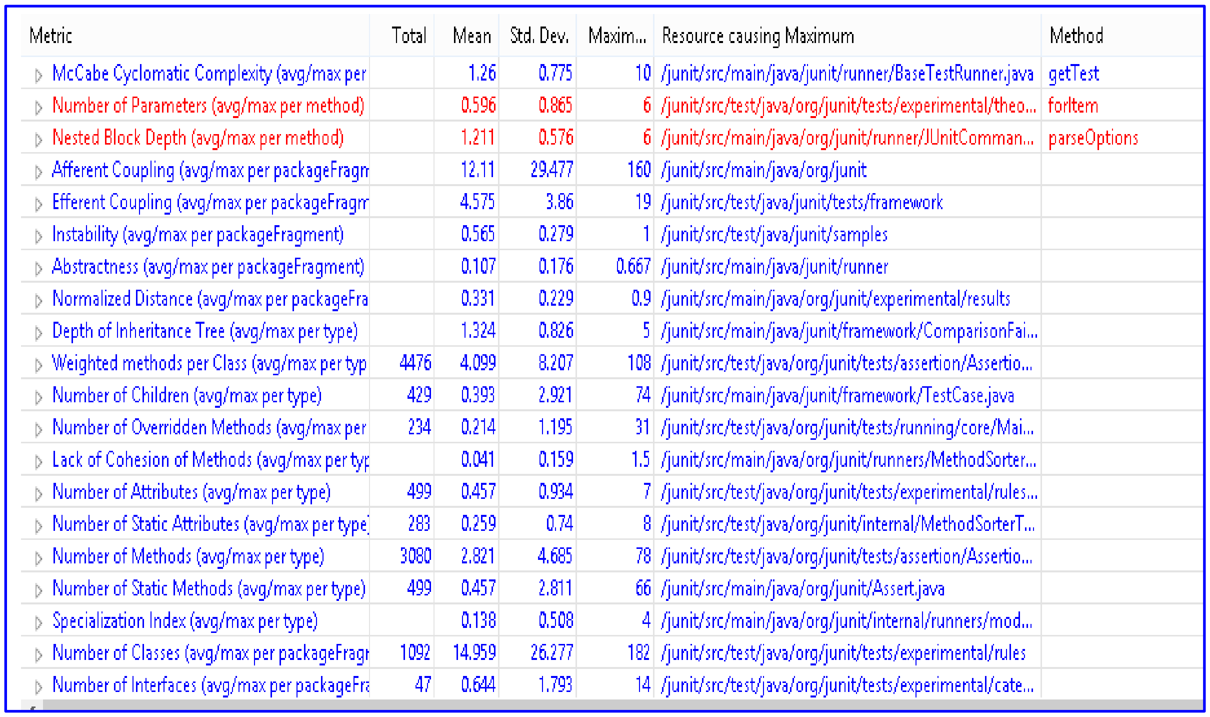
System	No. of classes	No. 0f Methods	Refactoring
Junit existing	1,267	4,124	553
Junit refactored	1,267	4,124	200

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.

4.2 EXPERIMENTAL RESULTS:

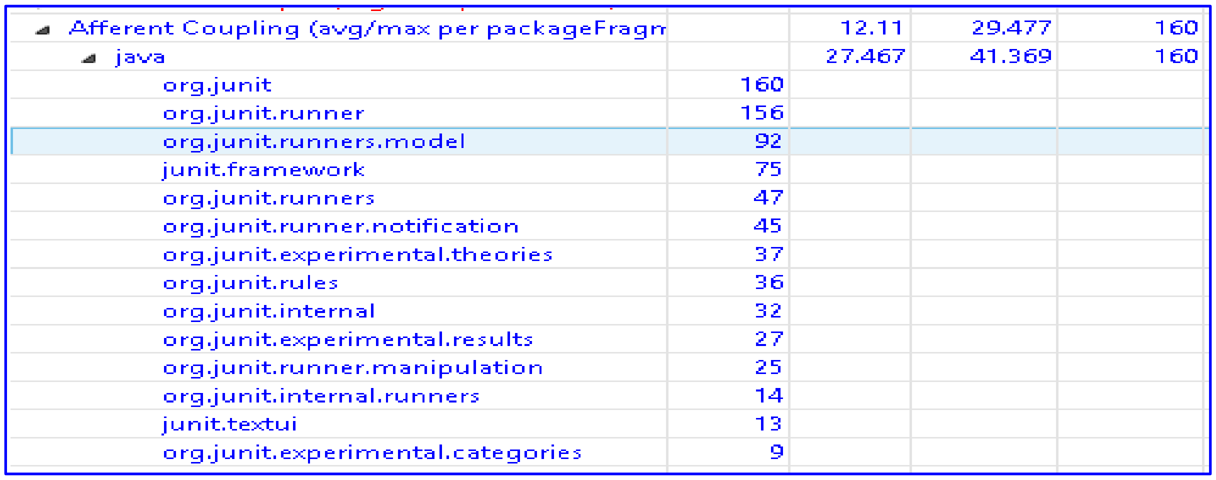
Result before applying proposed algorithm:



Metric	Total	Mean	Std. Dev.	Maxim...	Resource causing Maximum	Method
McCabe Cyclomatic Complexity (avg/max per		1.26	0.775	10	/junit/src/main/java/junit/runner/BaseTestRunner.java	getTest
Number of Parameters (avg/max per method)		0.596	0.865	6	/junit/src/test/java/org/junit/tests/experimental/theo...	forItem
Nested Block Depth (avg/max per method)		1.211	0.576	6	/junit/src/main/java/org/junit/runner/JUnitComman...	parseOptions
Afferent Coupling (avg/max per packageFragn		12.11	29.477	160	/junit/src/main/java/org/junit	
Efferent Coupling (avg/max per packageFragn		4.575	3.86	19	/junit/src/test/java/junit/tests/framework	
Instability (avg/max per packageFragment)		0.565	0.279	1	/junit/src/test/java/junit/samples	
Abstractness (avg/max per packageFragment)		0.107	0.176	0.667	/junit/src/main/java/junit/runner	
Normalized Distance (avg/max per packageFra		0.331	0.229	0.9	/junit/src/main/java/org/junit/experimental/results	
Depth of Inheritance Tree (avg/max per type)		1.324	0.826	5	/junit/src/main/java/junit/framework/ComparisonFai...	
Weighted methods per Class (avg/max per typ	4476	4.099	8.207	108	/junit/src/test/java/org/junit/tests/assertion/Assertio...	
Number of Children (avg/max per type)	429	0.393	2.921	74	/junit/src/main/java/junit/framework/TestCase.java	
Number of Overridden Methods (avg/max per	234	0.214	1.195	31	/junit/src/test/java/org/junit/tests/running/core/Mai...	
Lack of Cohesion of Methods (avg/max per typ		0.041	0.159	1.5	/junit/src/main/java/org/junit/runners/MethodSorter...	
Number of Attributes (avg/max per type)	499	0.457	0.934	7	/junit/src/test/java/org/junit/tests/experimental/rules...	
Number of Static Attributes (avg/max per type)	283	0.259	0.74	8	/junit/src/test/java/org/junit/internal/MethodSorterT...	
Number of Methods (avg/max per type)	3080	2.821	4.685	78	/junit/src/test/java/org/junit/tests/assertion/Assertio...	
Number of Static Methods (avg/max per type)	499	0.457	2.811	66	/junit/src/main/java/org/junit/Assert.java	
Specialization Index (avg/max per type)		0.138	0.508	4	/junit/src/main/java/org/junit/internal/runners/mod...	
Number of Classes (avg/max per packageFragr	1092	14.959	26.277	182	/junit/src/test/java/org/junit/tests/experimental/rules	
Number of Interfaces (avg/max per packageFra	47	0.644	1.793	14	/junit/src/test/java/org/junit/tests/experimental/cate...	

Figure4.1 List of Metrics for release Junit

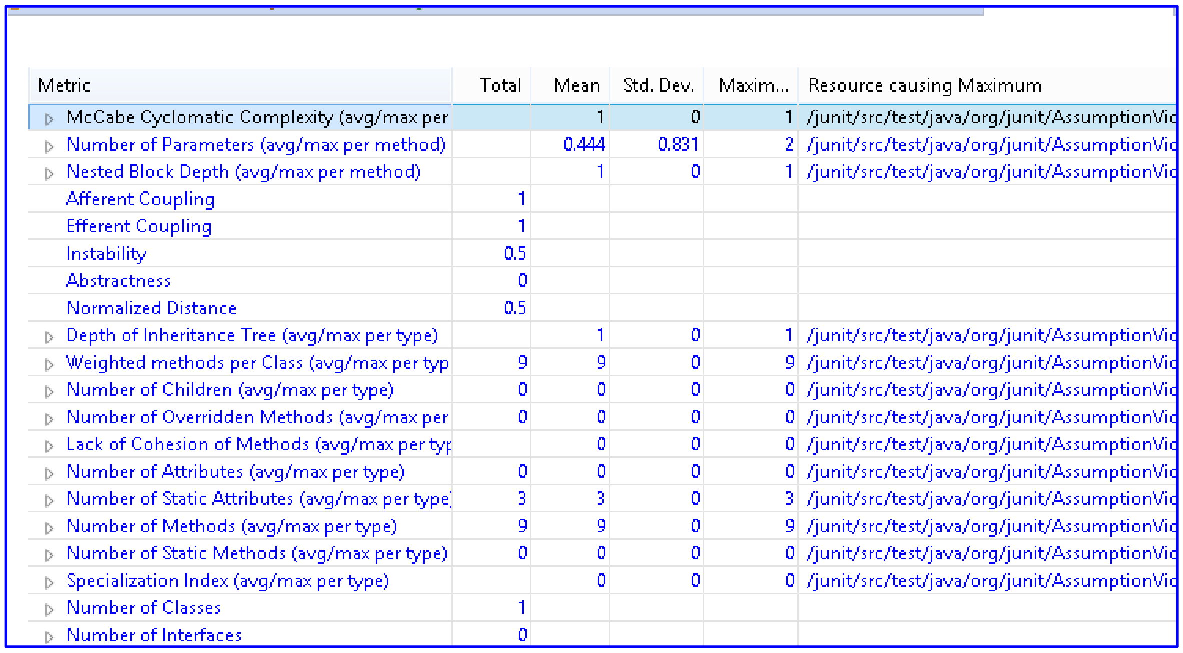
Individual metrics:



Afferent Coupling (avg/max per packageFragn		12.11	29.477	160
java		27.467	41.369	160
org.junit	160			
org.junit.runner	156			
org.junit.runners.model	92			
junit.framework	75			
org.junit.runners	47			
org.junit.runner.notification	45			
org.junit.experimental.theories	37			
org.junit.rules	36			
org.junit.internal	32			
org.junit.experimental.results	27			
org.junit.runner.manipulation	25			
org.junit.internal.runners	14			
junit.textui	13			
org.junit.experimental.categories	9			

Figure 4.2 Individual metrics

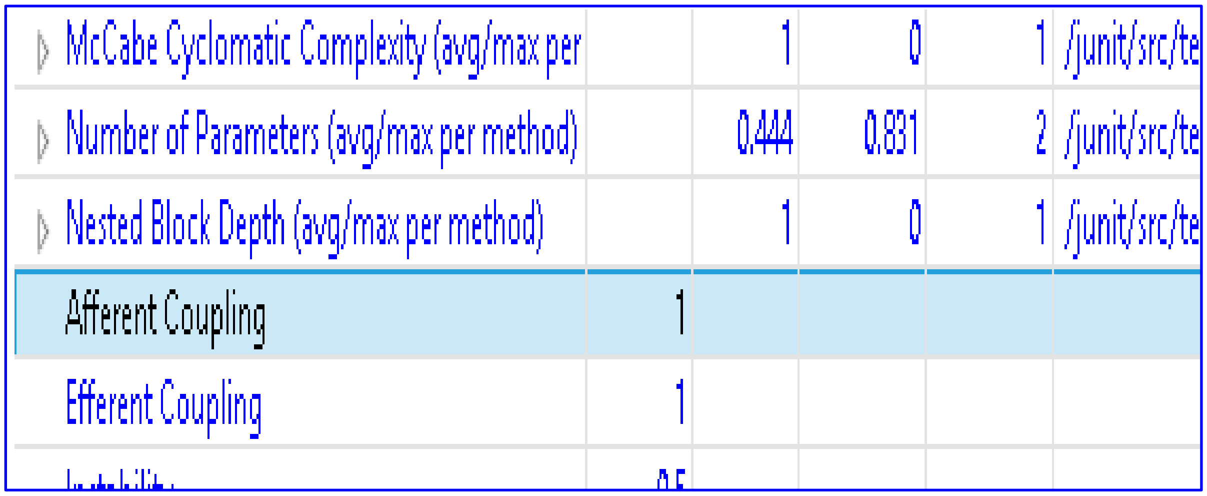
Result after applying proposed algorithm



Metric	Total	Mean	Std. Dev.	Maxim...	Resource causing Maximum
McCabe Cyclomatic Complexity (avg/max per		1	0	1	/junit/src/test/java/org/junit/AssumptionVi
Number of Parameters (avg/max per method)		0.444	0.831	2	/junit/src/test/java/org/junit/AssumptionVi
Nested Block Depth (avg/max per method)		1	0	1	/junit/src/test/java/org/junit/AssumptionVi
Afferent Coupling	1				
Efferent Coupling	1				
Instability	0.5				
Abstractness	0				
Normalized Distance	0.5				
Depth of Inheritance Tree (avg/max per type)		1	0	1	/junit/src/test/java/org/junit/AssumptionVi
Weighted methods per Class (avg/max per typ	9	9	0	9	/junit/src/test/java/org/junit/AssumptionVi
Number of Children (avg/max per type)	0	0	0	0	/junit/src/test/java/org/junit/AssumptionVi
Number of Overridden Methods (avg/max per	0	0	0	0	/junit/src/test/java/org/junit/AssumptionVi
Lack of Cohesion of Methods (avg/max per typ		0	0	0	/junit/src/test/java/org/junit/AssumptionVi
Number of Attributes (avg/max per type)	0	0	0	0	/junit/src/test/java/org/junit/AssumptionVi
Number of Static Attributes (avg/max per type)	3	3	0	3	/junit/src/test/java/org/junit/AssumptionVi
Number of Methods (avg/max per type)	9	9	0	9	/junit/src/test/java/org/junit/AssumptionVi
Number of Static Methods (avg/max per type)	0	0	0	0	/junit/src/test/java/org/junit/AssumptionVi
Specialization Index (avg/max per type)		0	0	0	/junit/src/test/java/org/junit/AssumptionVi
Number of Classes	1				
Number of Interfaces	0				

Figure4.3 List of matrices after applying refactoring technique

Individual Metrics:



McCabe Cyclomatic Complexity (avg/max per		1	0	1	/junit/src/te
Number of Parameters (avg/max per method)		0.444	0.831	2	/junit/src/te
Nested Block Depth (avg/max per method)		1	0	1	/junit/src/te
Afferent Coupling	1				
Efferent Coupling	1				

Figure 4.4 Individual matrices

4.3 COMPARISION WITH EXISTING TECHNIQUE

In this section we evaluated results of the gathered refactoring dataset as for programming maintainability. In first case, we define the consequences of the examination on the practicality of refactored classes. A while later, we present the discoveries on the effect of refactorings on source code estimations and upgrading the RMI index there by reducing refactoring requirement on build on build basis in Junit dataset.

4.3.1 Metrics change: The graph below shows ratio of change in metrics after applying refactoring. The refactoring techniques move method and mode, move field, extract class are applied to Junit releases 4.10, 4.11, 4.12 and 5.0 releases to compare the effectiveness of our approach for refactoring. Impact of method extraction on release based on logic breaking helps in reducing WMC.

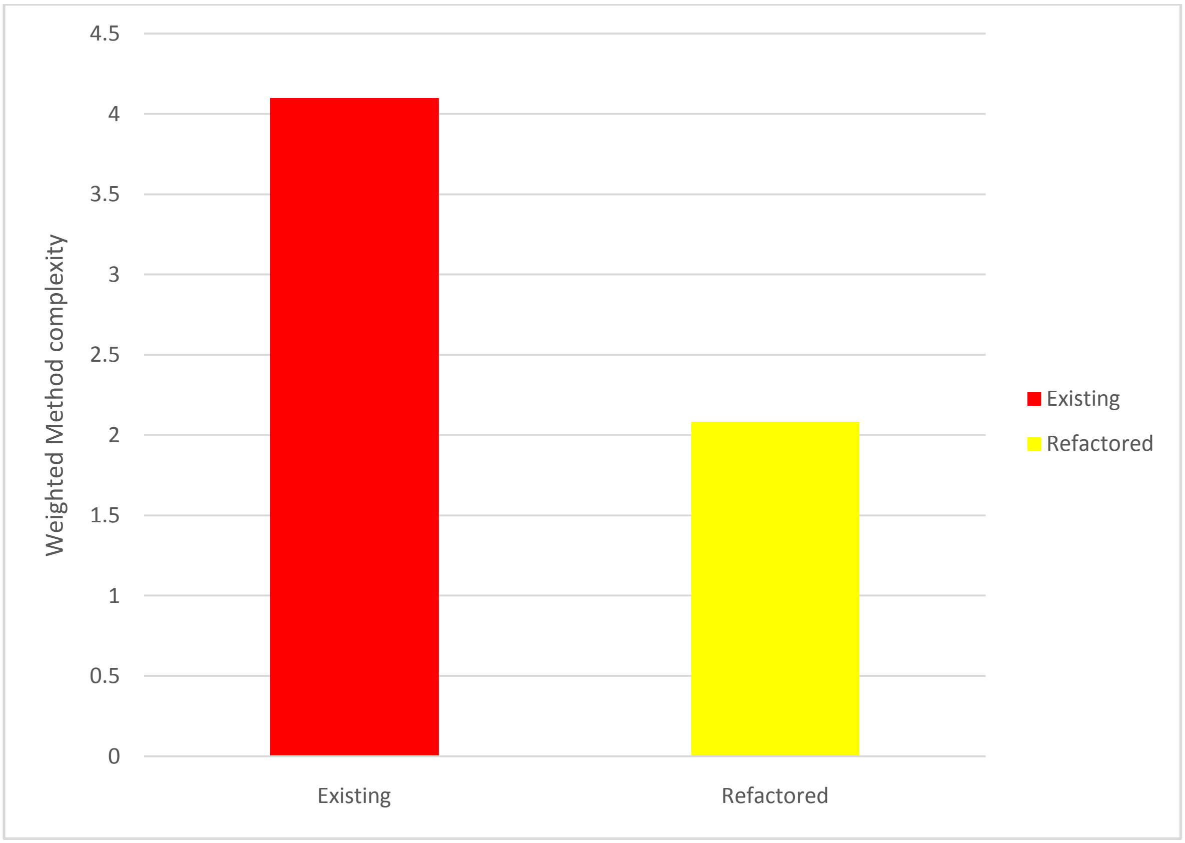


Figure4.5 Metrics Change

4.3.2 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.

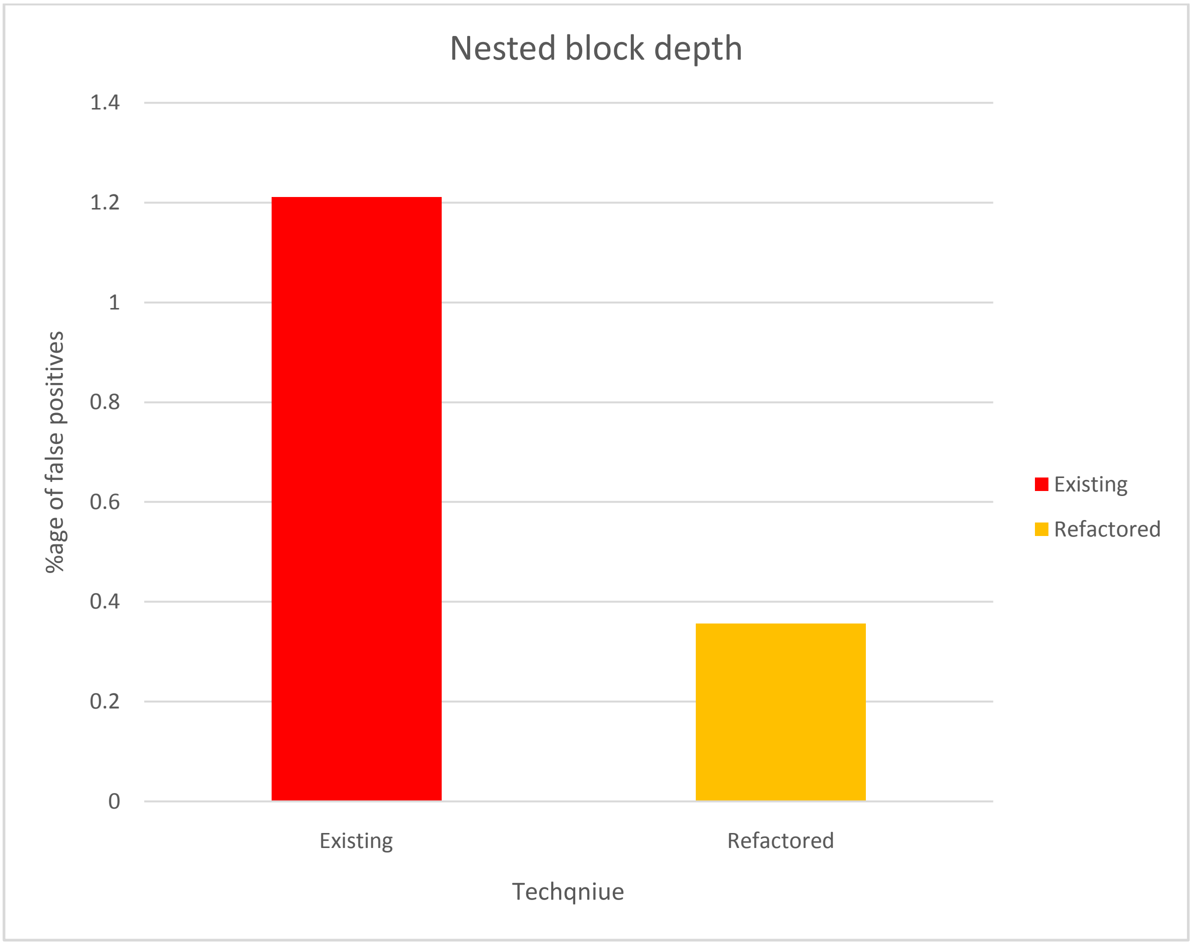


Figure4.6. Nested block depth

4.3.3 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.

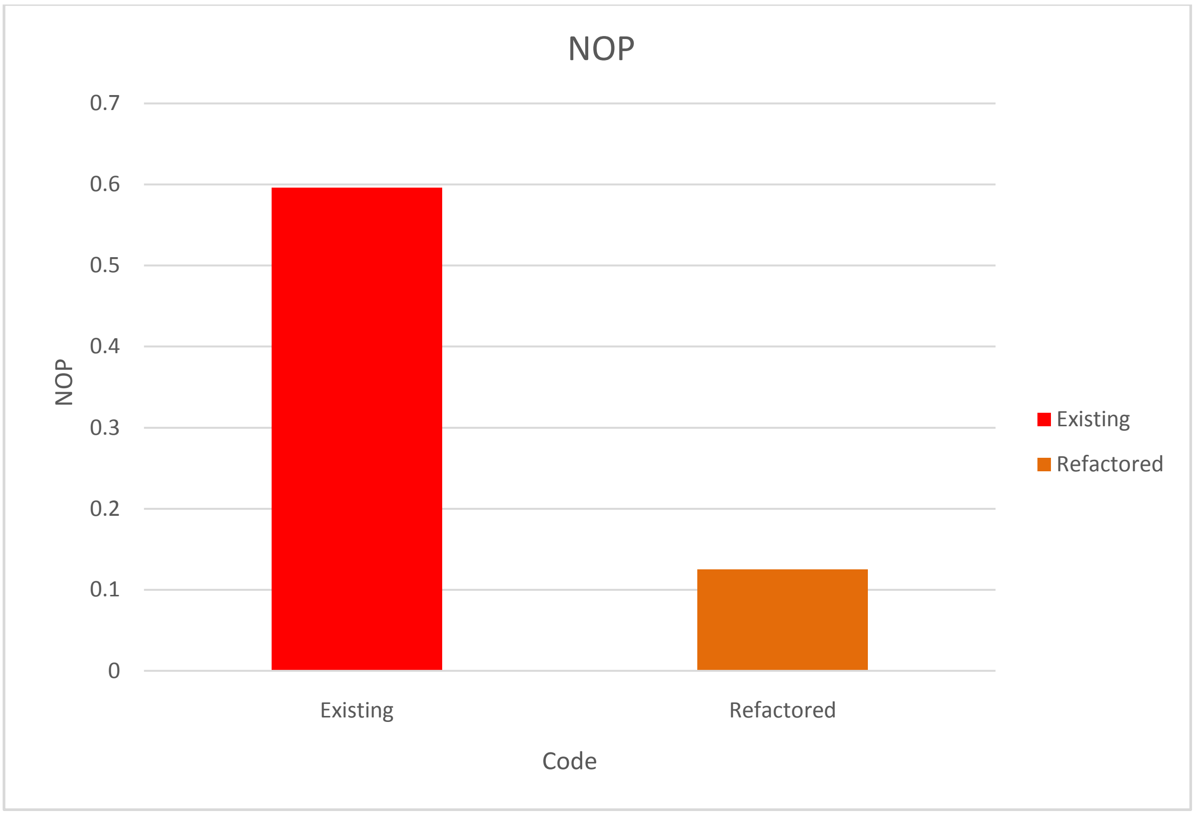


Figure4.7. Number of parameters

Table 1.4 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.

Table1.5. Metrics improvement

System name	CI	WMC	NOI	RFC	TCLOC	TLLOC	TNOS
Enhanced							
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

4.3.4 Number 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.

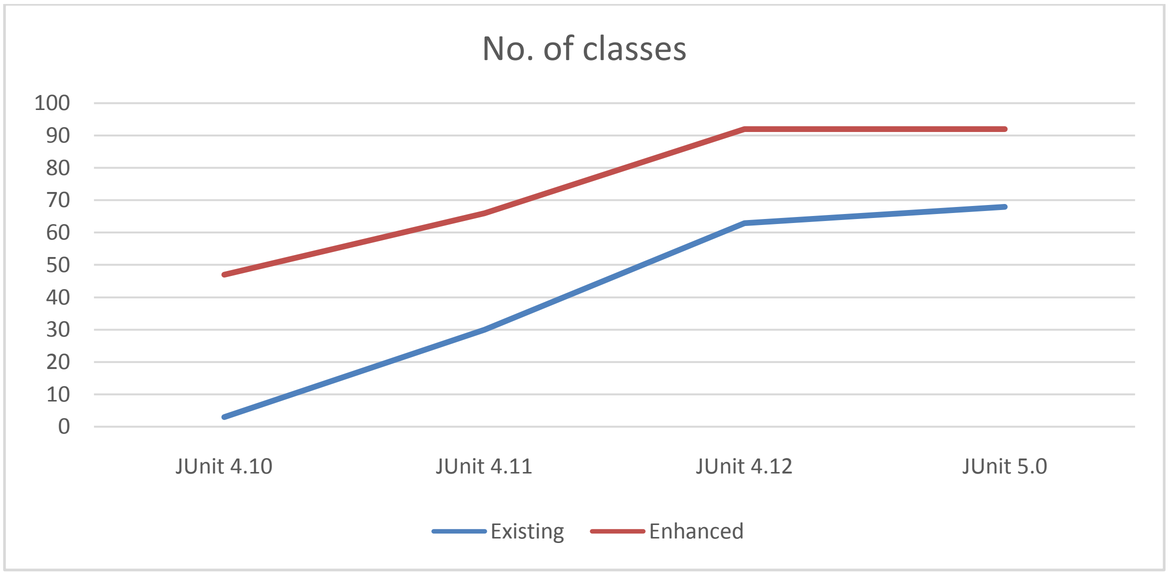


Figure4.8.Number of classes

4.3.5 Number of Interfaces: Interfaces would increase with increase in inheritance.

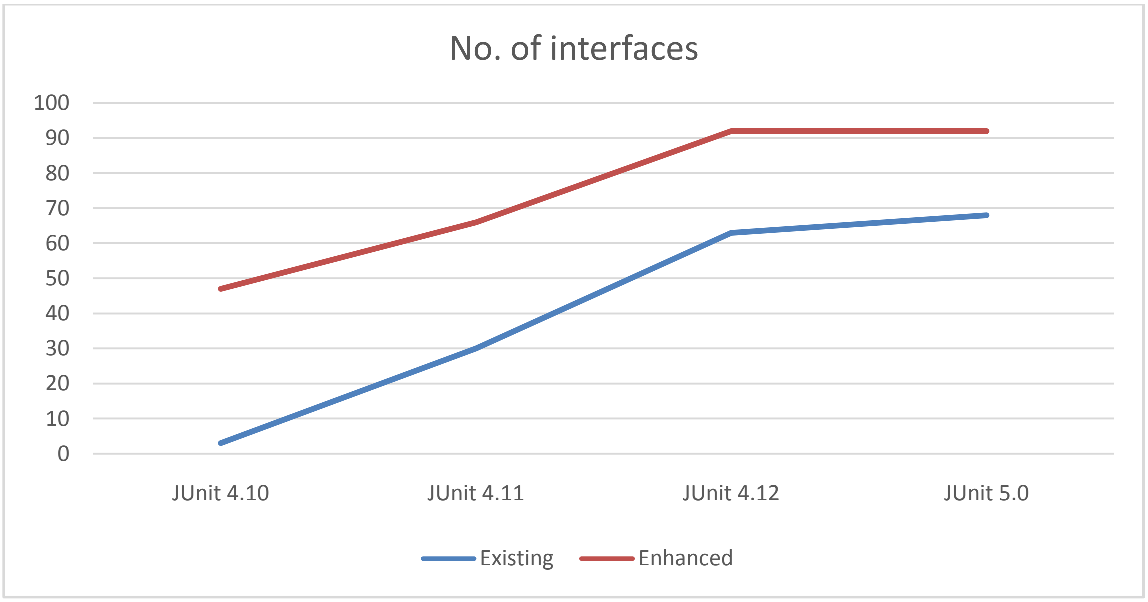


Figure4.9.Number of Interfaces

CHAPTER 5

CONCLUSION AND FUTURE SCOPE

5.1 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.

5.2 FUTURE SCOPE: 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.

REFRENCES

[1] I. Kádár and P. Heged, “A Code Refactoring Dataset and 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.

[39] T. Mens and T. Tourwe, "A Survey of Software Refactoring," vol. 30, no. 2, pp. 126–139, 2004.

[40] D. Cedrim and A. Garcia, "Does refactoring improve software structural quality ? A longitudinal study of 25 projects," no. i, pp. 73–82.

[41] D. Chatterji, J. C. Carver, N. A. Kraft, and J. Harder, "Effects of Cloned Code on Software Maintainability : A Replicated Developer Study," pp. 112–121, 2013.

[42] M. Abebe and C. Yoo, "Trends , Opportunities and Challenges of Software Refactoring : A Systematic Literature Review," vol. 8, no. 6, pp. 299–318, 2014.

[43] L. Tahvildari and K. Kontogiannis, "Improving design quality using meta-pattern

transformations : a metric-based approach," vol. 361, no. October 2003, pp. 331–361, 2004.

[44] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, and E. Figueiredo, "A Review-based Comparative Study of Bad Smell Detection Tools," no. Dcc, 2016.