Investigating the Effects of Refactoring on Software Maintainability

A dissertation submitted in the partial fulfillment for the award of Degree of

Master of Technology

In

Software Technology

by

Sachin Gaur (Roll no. 2K11/ST/17)

Under the guidance of

Dr. Ruchika Malhotra

DEPARTMENT OF SOFTWARE ENGINEERING

DELHI TECHNOLOGICAL UNIVERSITY

BAWANA ROAD, DELHI

2014

DECLARATION

I hereby want to declare that the thesis entitled "**Investigating the Effects of Refactoring on Software Maintainability"** which is being submitted to the **Delhi Technological University**, in partial fulfillment of the requirements for the award of degree in **Master of Technology in Software Technology** is an authentic work carried out by me. The material contained in this thesis has not been submitted to any institution or university for the award of any degree.

_____

Sachin Gaur

Department of Software Engineering

Delhi Technological University,

Delhi.

CERTIFICATE

DELHI TECHNOLOGICAL UNIVERSITY

BAWANA ROAD, DELHI-110042

Date: _____

This is to certify that the thesis entitled "**Investigating the Effects of Refactoring on Software Maintainability"** submitted by **Sachin Gaur (Roll Number: 2K11/ST/17),** in partial fulfillment of the requirements for the award of degree of Master of Technology in Software Technology, is an authentic work carried out by him under my guidance. The content embodied in this thesis has not been submitted by him earlier to any institution or organization for any degree or diploma to the best of my knowledge and belief.

Dr. Ruchika Malhotra ,

Asst. Professor, Department of Software Engineering ,

Delhi Technological University, Delhi-110042

ACKNOWLEDGEMENT

I would like to take this opportunity to express my appreciation and gratitude to all those who have helped me directly or indirectly towards the successful completion of this work.

Firstly, I would like to express my sincere gratitude to my guide **Dr. Ruchika Malhotra**, **Assistant Professor, Department of Software Engineering**, **Delhi Technological University, Delhi** whose benevolent guidance, encouragement, constant support and valuable inputs were always there for me throughout the course of my work. Without her continuous support and interest, this thesis would not have been the same as presented here.

Besides my guide, I would like to thank **Mr. Nakul Pritam**, PhD. Scholar DTU for his valuable suggestions. I would also like to thank my wife, **Ms. Shikha Sharma** for her continuous motivation and understanding. Also I would like to extend my thanks to the entire staff in the Department of Software Engineering, DTU for their help during my course of work.

SACHIN GAUR

ST 2K 11/ 17

ABSTRACT

Software evolution is a term used for repeated modifications in a software system caused by changing existing requirements, emerging new requirements or bug fixes. Refactoring is a process where a code is restructured is such a manner that certain attributes of the code improve, without having any effect on its external behavior. It improves the code design, making it easier to be understood and to be extended and it becomes quicker if any complicated feature or system is to be accommodated in it. The internal structure is improved. It removes bad smells from the code, which essentially means getting rid of unclear, duplicate or complicated design problems.

There are a number of Refactoring techniques available. It has been observed that different techniques have a different effect on various quality attributes of the code. Because each technique has a varied effect on various codes, it is difficult for a designer to decide which technique to opt for to get the desired effect. Each technique essentially has a different purpose and effect.

This study intends to generate a heuristics on these techniques. This classification will be based on the measurable effects that these Refactoring techniques have on various software quality attributes. These heuristics will help the designer develop an understanding as to which technique will change their code attributes in what manner and hence it will be easy for them to predict the change that their software quality will undergo after a particular Refactoring technique is applied to it.

In this study, we take into consideration the effect of Refactoring on four software quality attributes of software. This research focuses on presenting refactoring heuristics based on their quantifiable outcome on software quality attributes which takes into account its internal and external quality attributes. Also we have taken into consideration forty three Refactoring techniques out of all those suggested by Martin Fowler in his catalog [1], in order to study their effects on four software quality attributes namely Effectiveness, Flexibility, Reusability and Extendibility and thus affecting the overall maintainability. Also QMOOD quality model has been used to relate design measures to software quality attributes.

After a detailed logical analysis of these effects a heuristic has been generated which suggests that which refactoring technique improves a specific software quality and which one is susceptible to deteriorate it. Also which refactoring technique has no effect on it. Such heuristic results have been validated by an industrial survey. The refactoring heuristics deduced from our study was shared with a team of developers who work in real world industrial environment. They were asked to pick and apply the refactoring technique as suggested by this study and share his / her views on the same. They were then asked to fill up a survey form which pertained to the changes in quality of the software after applying refactoring. The results received in this survey are mostly in line with the observed heuristic data.

Overall, it is sure to help design engineers who have a particular design objective in their mind to pick the most suited Refactoring technique which will drift the quality attributes of their design towards the desired value. When any quality attribute needs to be changed then the most fitting Refactoring technique can be applied.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

INTRODUCTION

Every developer would agree with two facts: first it is extremely difficult to write a perfectly designed code in the first shot and secondly with changing times, the requirements from the software change and thus the software quality needs to be modified. Here comes in Refactoring. This activity of refactoring a code is basically making a succession of small transformations in the code so that the internal quality of the code changes without disturbing its external behavior. It is believed that Refactoring can positively impact the quality factors of a software like its effectiveness, Reusability, Flexibility and so on. Refactoring is not a magic wand and does all this improvement to the code by taking care of the bad smells of the code, by eliminating bad bugs and by fixing flaws and getting rid of the irregularities of the code. There is a very vast pool of Refactoring techniques available that can be intelligently used by developer to achieve their design goals with minimum effort and risk. These kind of changes are required to keep the software fit for the long run, to maintain it for a longer time. Also this will keep the code from decaying. Also its complexity stays within human acceptance range. There are a number of instances available that confirm that refactoring is done in real world industrial environments for effective quality up gradation. But it has also been observed that actually there is no definite comprehensive study available which empirically shows a relation between refactoring and its impact on software quality of a system. In our study here, we have attempted to address this problem of developers. We

intend to put forward a refactoring heuristic which will guide software developers regarding their choice of a Refactoring technique when they are looking to improve a particular software quality. This Refactoring heuristic will not only suggest the most suitable refactoring technique but will also warn them about those Refactoring techniques which might actually cause some quality factor to deteriorate.

1.1. SOFTWARE EVOLUTION AND REFACTORING

Every developer faces a situation where he needs to have some changes in the software. These changes are constantly required to generate a more suitable and cleaner code as compared to the existing one. But while doing this it is extremely important to avoid introducing any new bugs and there should be no side effects of the changes being made. This is where refactoring techniques come in. These techniques ensure that required changes are done with minimal introduction of bugs. As Fowler says [1] that refactoring is essentially a procedure where the attempt is to bring about a positive change in the internal behavior of the code, while taking care that these changes do not alter its external behavior at all. While refactoring we do a number of small changes in the code and the cumulative effect of these small changes can bring radical changes in the design of the code. Kent Beck gave a term "bad smell" for all the flaws in the software that stop it from qualifying as good quality software. Some examples of such bad smell can be:

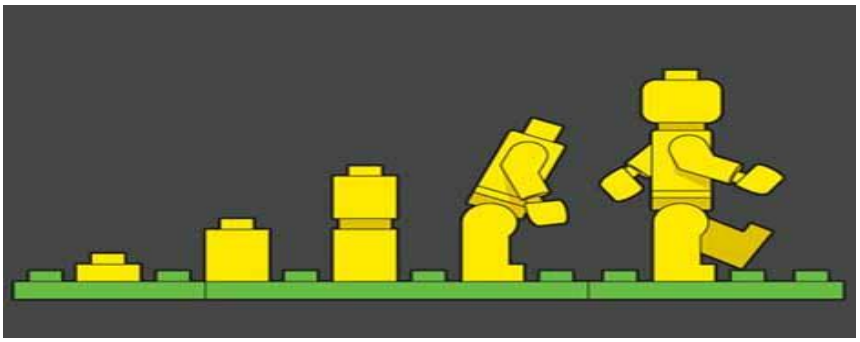
- Duplicate code. This means that the same change has to be done at a number of places.
- Bad organization of classes and methods. They can be either too small or too big.
- Loose coupling between structures.
- Too much or too little delegation.

Various Refactoring Techniques are used to remove such bad smells and introduce and enhance good qualities in software. Refactoring has its own advantages such as

- It ensures that the code does not lose its structure because of being constantly changed by the programmers (who generally do not know the complete design objective)
- The readability of the code is enhanced. This is good both for the user and he programmer himself. The intention of the code is conveyed in an effective manner.
- Refactoring makes it easy to introduce new functions down the line and to maintain it.
- It removes bad smells in a very controlled manner, with no side effects.

A software has to regularly evolve to stay useful. New functionalities are regularly added to the existing software to make it satisfying for new requirements.

1.2. Motivation of the work



The need for Refactoring has its roots in the four fundamental principles of Software design. These fundamental principles are as follows:

a. ***Improving the existing design.*** This implies that the improving the following parameters - reliability, portability, efficiency, maintainability and usability.

b. ***Decreasing the complexity.*** This implies that testing activities should be started early and move parallel with the development of software. Thus, test case prioritization should focus on prioritizing the test cases on the basis of requirement specification.

c. ***Minimize the Code duplication and redundancy.*** This implies that code duplication which usually occurs from copy-and-modify operations.

d. ***Reduce the development time and make the process of Maintenance and Evolution, simple and effortless.***

This simply means that we develop a code that is of high quality and is well factored. A code that is easy to be maintained and if there is a requirement then it can be extended with minimum difficulty. They say that a good program is one that can be easily understood by humans & not just computers, and we support that. Many changes to the system are now easier to make because they have smaller impact and it's more obvious how to make the appropriate changes.

1.3. Goals of this thesis

The goal of the work in this thesis is summarized below:

a. ***To apply the various Refactoring techniques on User interface framework code of mobile software -*** As discussed earlier, refactoring intends to increase the quality of existing software but it can lead to change in existing design. There are a number of refactoring techniques proposed with few of these orthogonal to each other e.g. Extract Class is

orthogonal to Inline Class. Applying these to Complex Mobile software code which is multithreaded based on android platform.

b. ***To validate the proposed heuristic and measure various parameters before and after refactoring of code-*** We also aim to validate the proposed heuristic and calculate different metrics like Design Size, Abstraction etc. The results derived from these are again compared by survey done by a team of developers, so we wish to generalize the results.

c. ***To analyze the change in metrics due to refactoring and find correlation between actual heuristic results and developer opinion-*** We also try to correlate the results shown by heuristics, and actual developer's opinion. We aim to analyze the results for effectiveness in terms of user effectiveness.

It can be observed that our goals are focused on improving the quality of existing design and help designers and code reviewers to improve the metrics parameters to increase the overall quality of code.

1.4. Organization of thesis

This thesis is organized as follows:

***Chapter* 2** discusses the previous work done in the field of measuring various software attributes after the application of refactoring techniques. This includes the extensive study of various refactoring techniques and measurements that have been proposed in the literature so far. It also highlights some of the most relevant works in the direction of field of work presented in the thesis

Chapter 3 gives a comprehensive study of Refactoring Techniques and their pros / cons. This chapter is dedicated to a profound study of historical background of Refactoring including

details of its origin. We also exemplified the working of these techniques with some sample data.

Chapter 4 is about research methodology used herein. It describes the objectives of this research and also the quality model used for the same. It shows how refactoring changes various quality matrices of a software which in turn affects the software quality attributes, both internal and external. It also defines the various design metrics which are considered. And finally it gives us the effects of refactoring techniques on quality factors.

Chapter 5 In this chapter we analyses the impact of refactoring techniques on various software qualities under consideration. We derive heuristics which establish a relationship between the refraction techniques and the software qualities of a code.

Chapter 6 In this chapter we give our final concluding remarks. Also we mention the limitations of this study and what are the plans for any future study.

CHAPTER 2

LITERATURE SURVEY

Any serious software developer would agree with the fact that software codes are in need of modification all the time. Blame it on changing requirements or environment, changes need to be done in codes to enable it to get modified easily and quickly. A software developer has to make changes in the internal structure of the software design to make it compatible for getting modified and suit the new set of requirements. This is done by Refactoring the code.

Refactoring is a powerful tool that can help a developer to achieve the preset goal but only if it is done in the right way and carefully. There are all the possibilities that Refactoring can introduce new bugs into your code or might have some un intended consequences that are not so good. A developer who is Refactoring has to be sure that none of this happens or else he is digging a hole for himself to fall into. So basically Refactoring needs to be done carefully and systematically, avoiding any mishaps like introduction of a subtle bug into it.

The first emergence of the Refactoring activities can be dated in the early 90s and they say that it was probably coined initially in Smalltalk. But soon enough it gained the interest of the entire software community, probably because it worked. Refactoring has its own benefits like:

- Refactoring ensures that the design will not decay with time
- The design of the software improves on the whole

- The code becomes easier to understand for a new developer who knows nothing about it.
- In the process of Refactoring, one is sure to stumble upon some dormant bugs that the code has and can remove them.
- Refactoring ensures that you have no bad smells in the code.
- When a code is rendered cleaner and simpler by Refactoring it becomes easier to make changes in it in future. There have been instances where huge projects have failed only because the code was cluttered and not clean.

In this section we try to present a summary of all those empirical studies that have been conducted so for to analyze the effect of Refactoring a code on its quality factors. Some of the notable works are as follows:

- Bansiya and Davis have shown in their study [2] that it is possible to have an empirical relation between software internal properties and refactoring techniques.
- Meyer in his book [3] shows that there is a general understanding that when you improve the design properties of your software then the quality factors like functionality , reusability, understandability and efficiency are bound to show some improvement.
- T. Mens et all showed in their study [4], how to categorize refactoring activities on the basis of the software quality which is finally improved on their application.
- Sahraoui et al, in their paper [5] have shown that there are a number of refactoring techniques that should be preferred to other if a developer is looking at improving the maintainability of his software. To come to this conclusion they used the

design properties of coupling and inheritance and made some empirical rules for them.

- R.V.Kapoor and E. Stroulia presented a paper in a conference [6] where they showed that the software quality of Extendibility of a code can be changed if we did refactoring. This happens because refactoring reduces the design measures of coupling and size.
- A paper by Y. Kataoka, T . Imai and others [7] gave validated results for showing that Refactoring can positively boost up the maintainability of a code. They used a C++ program to validate these findings for two refactoring techniques.
- At the Ninth Metrics symposium, R. Leitch along with E. Stoulia presented their extensive study [8] of the advantages of design restructuring focusing on the maintainability of the code. They also took into consideration of the cost factor of maintaining the code and how it reduces when we refactor. They considered two systems which had bad smells and to fix them two refactoring techniques, namely Move Method and Extract Method were used. They finally concluded that it was possible to improve the quality of the code by doing so and this happened because refactoring reduced the density of dependencies and size and increased the procedure count.
- A study done by B.D.Bois, S.Demeyer and J.Verelst [9] is about the improving two important measures of any code, its Coupling and Cohesion. They suggested refactoring techniques that will improve these two measures and also went on to guide the developer about those refactoring techniques which would not be useful at all for this task and even those techniques which might deteriorate the cohesion and coupling of a system. Thus an optimized approach was suggested. They

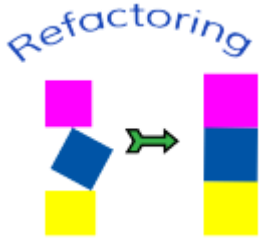
proved that their findings were good enough by applying the relevant results to refactor Tomcat by Apache.

- J Ratinzer, M. Fisher and H. Gall have done a lot of work in the field of refactoring and in one of their papers [10] they have worked on the evolvability of the code. Their study throws light upon the fact that by refactoring a code we can reduce the change coupling among program's source code files.
- In 2009 M.Alshayeb presented a study [11] wherein effects of refactoring on software qualities were investigated and empirical relations were derived. These relations depicted some negative impact of refactoring techniques too.

This is just a glimpse of the studies done regarding refactoring and its impact on different quality factors. But we see here that most of the studies performed have been done taking into consideration only a couple of refactoring techniques. Also, the software quality factors in each study have not been more than two. Hence when this study was planned it was decided to pick up a larger pool of refactoring techniques and to consider more number of design properties. Here it has been tried to present a more comprehensive and elaborate heuristics for guiding the process of software development and refactoring.

CHAPTER 3

REFACTORING TECHNIQUES



If one follows Fowler [1], he gives us a list of twenty two bad smells that a code might have like, duplicate code, Large class, divergent change, Data clumps, speculative generality, lazy class, middle man and so on. He also gives a long list of refactoring activities that can be used to fix these smells and defects in the code. Also according to Fowler [1], these Refactoring Techniques can further be categorized under six different categorize. We take a few examples from each category in the following part of this chapter.

3.1 Composing Methods

1. **Extract Method**: A method is taken out from a lengthy method to have short methods. Short methods increase re-usability that improves readability and understandability.

```
void printOwing() {
  printBanner();

  //print details
  System.out.println ("name:  " + _name);
  System.out.println ("amount " + getOutstanding());
}
```

```
void printOwing() {
  printBanner();
  printDetails(getOutstanding());
}

void printDetails (double outstanding) {
  System.out.println ("name:  " + _name);
  System.out.println ("amount " + outstanding);
}
```

Figure 1: Extract Method

2. **Inline Method**: This technique is the opposite for extract method. A group of badly factored methods can be put right by this technique. This method can also be used to get rid of useless and too much indirection or when one is getting lost in delegation.

```
int getRating() {
  return (moreThanFiveLateDeliveries()) ? 2 : 1;
}
boolean moreThanFiveLateDeliveries() {
  return _numberOfLateDeliveries > 5;
}
```

```
int getRating() {
  return (_numberOfLateDeliveries > 5) ? 2 : 1;
}
```

Figure 2: Inline Method

3. **Replace Method with Method Object**: A lengthy method that uses many local variables that makes extract method technique difficult to apply. What we do is that we turn all the local variables into fields on that object and to do so we convert method into its own object. This makes the code more comprehensive.

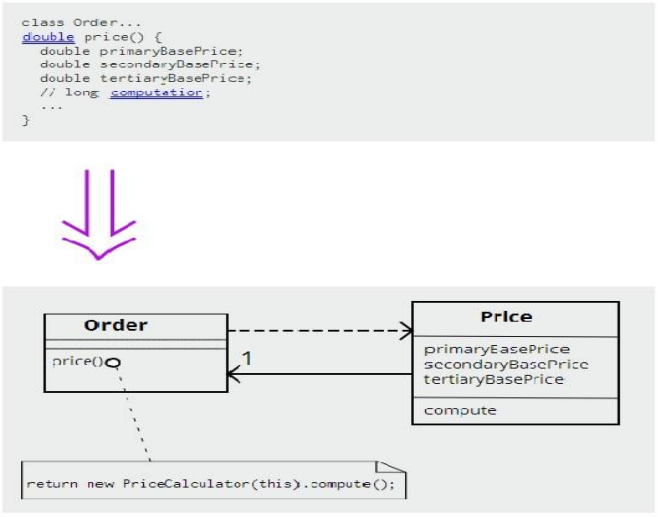


Figure 3: Replace Method with Method Object

3.2 Moving Features Between Objects

1. **Move Method**: A method is used by more features of another class than the class where it exists. Here we change the original method into a simple delegation or we get rid of it completely.

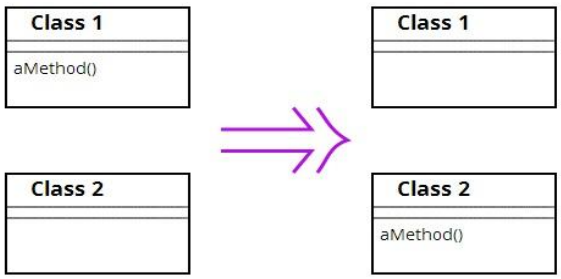


Figure 4: Move Method

2. **Move Field**: A field which is defined on a class is used by another class more than the class where it originally is present. What it does it that it moves that field from its original class to the fields whose methods are using it more than the existing class methods.

3. **Extract class**: A class does performs the job of two or more classes. We create a separate new class and shift the relevant attributes and methods to the new class.

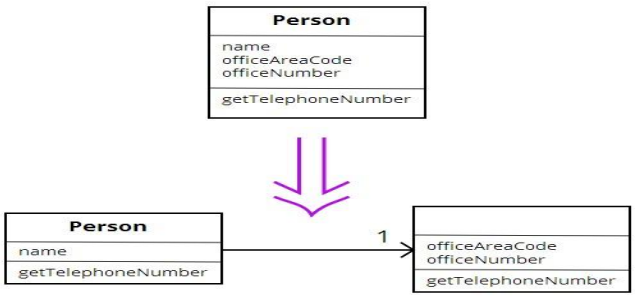


Figure 5: Extract Class

4. **Inline Class**: This technique is the opposite of Extract class. Here we have a class that is doing very less and hence we move its features to some other class and get rid of it.

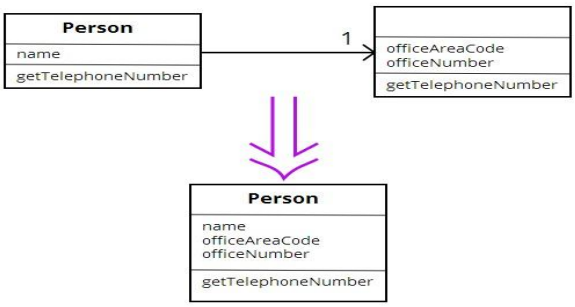


Figure 6: Inline Class

3.3 Organizing Data

1. **Replace Data Value with Object**: A data item needs additional data or behavior. Encapsulate the data item in its own object.

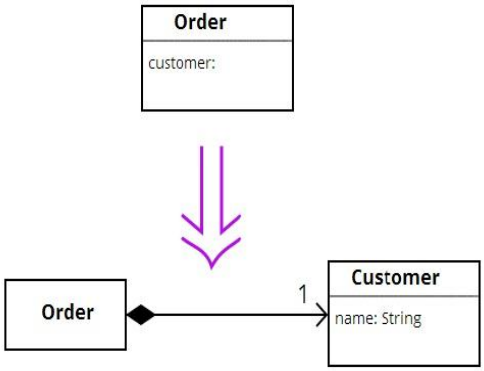


Figure 7: Replace Data Value with Data

2. **Replace Array with Object**: We have an array that contains certain elements that mean dissimilar things. Change the array with an object which has a field for every element.

Figure 8: Replace Array with Object

3.4 Simplifying Conditional Expressions

1. **Replace Conditional with Polymorphism**: We have a conditional that chooses diverse actions depending on the type of the object. Shift each part of the conditional to an overriding function in a subclass. Create the original method abstract.

2. **Decompose Conditional**: We have a complex conditional (if-then-else) statement. Derive methods from the condition to simplify.

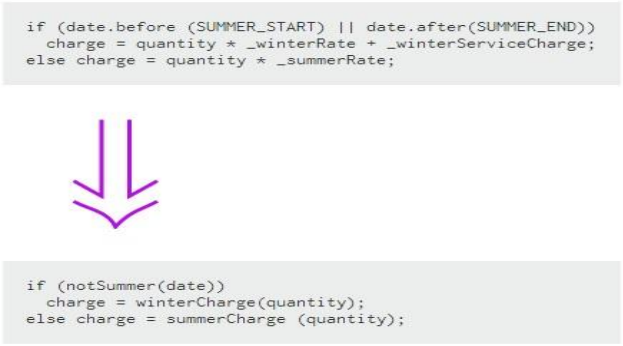


Figure 9: Decompose Conditional

3.5 Making Method Calls Simpler

1. **Rename Method**: The name of a function doesn't tell its purpose. Change the name of the method to make it more user friendly.

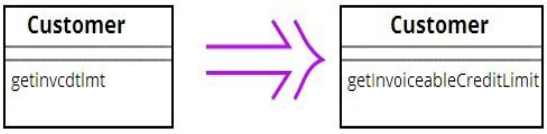


Figure 10: Rename Method

2. **Replace Parameter with Method**: A class object calls a method and passes the result as a parameter for a method. The recipient can also call this method. Remove this parameter and let the receiver call the method.

Figure 11: Replace Parameter with Method

3.6 Dealing With Generalization

1. **Pull Up Method**: We have methods with identical results on subclasses. Move them to the super class.

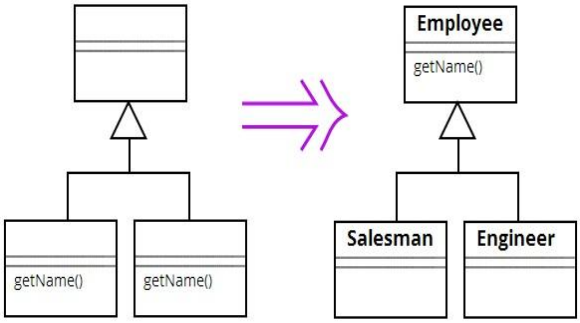


Figure 12: Pull Up Method

2. **Push Down Method**: Activities on a superclass is relevant only for some of its subclasses. Shift it to those subclasses.

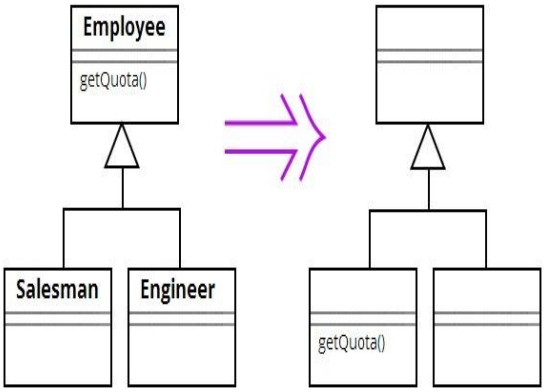


Figure 13: Push Down Method

3. **Push Down Field**: A field is used only by some subclasses. Shift that field to those subclasses.

4. **Extract Sub Class**: A class has features that are used only in some object instances. Make a subclass for that subset of features.

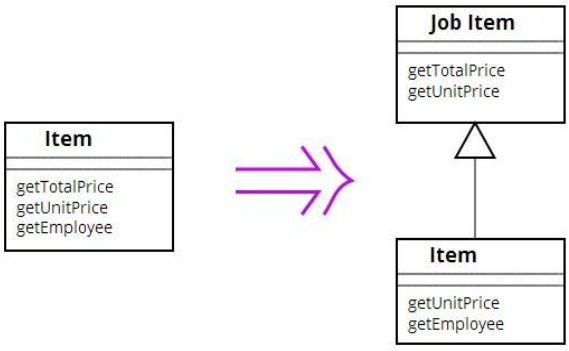


Figure 14: Extract Sub Class

5**. Extract Super Class**: We have two classes with almost similar features. Make a super class and shift all the common features to the super class.

6. **Extract Interface**: Several objects use the same subset of a class's interface, or two classes have common interfaces partly. Extract that subset into an interface.

CHAPTER 4

RESEARCH METHODOLOGY

4.1 Objectives

We are going to concentrate on three major points in this thesis:

1. We know that each software code has its own quality factors to define its quality. Whenever we need to bring about any changes in the quality of a code we need to refactor it. Here in our study we have chosen a few refactoring techniques which have been given by Fowler [1] in his catalogue and we are studying the effects of these techniques on some chosen quality factors. Now we need a quality model that will give us a relation between various internal and external quality attributes of a software and its design properties. So here we use QMOOD, a comprehensive model, suggested by Bansiya and Davis [2]. Using it we define a relation between a codes design properties and its quality factors. Thus we deduce relations between refactoring and software quality attributes.

2. Next we deduce a heuristics of refracting techniques that can work as a guide for developers who are aiming for some goal oriented refactoring and are looking for improving definite quality factors of a software code. This heuristics will give them a definitive idea as to which refactoring technique will be suitable and which one will be not.

3. Next it is important to find out if the findings of this study are in sync with the refactorings in real context. So finally we validate our heuristics.

The research Methodology followed here in this study can be summed up as the flow chart shown below:

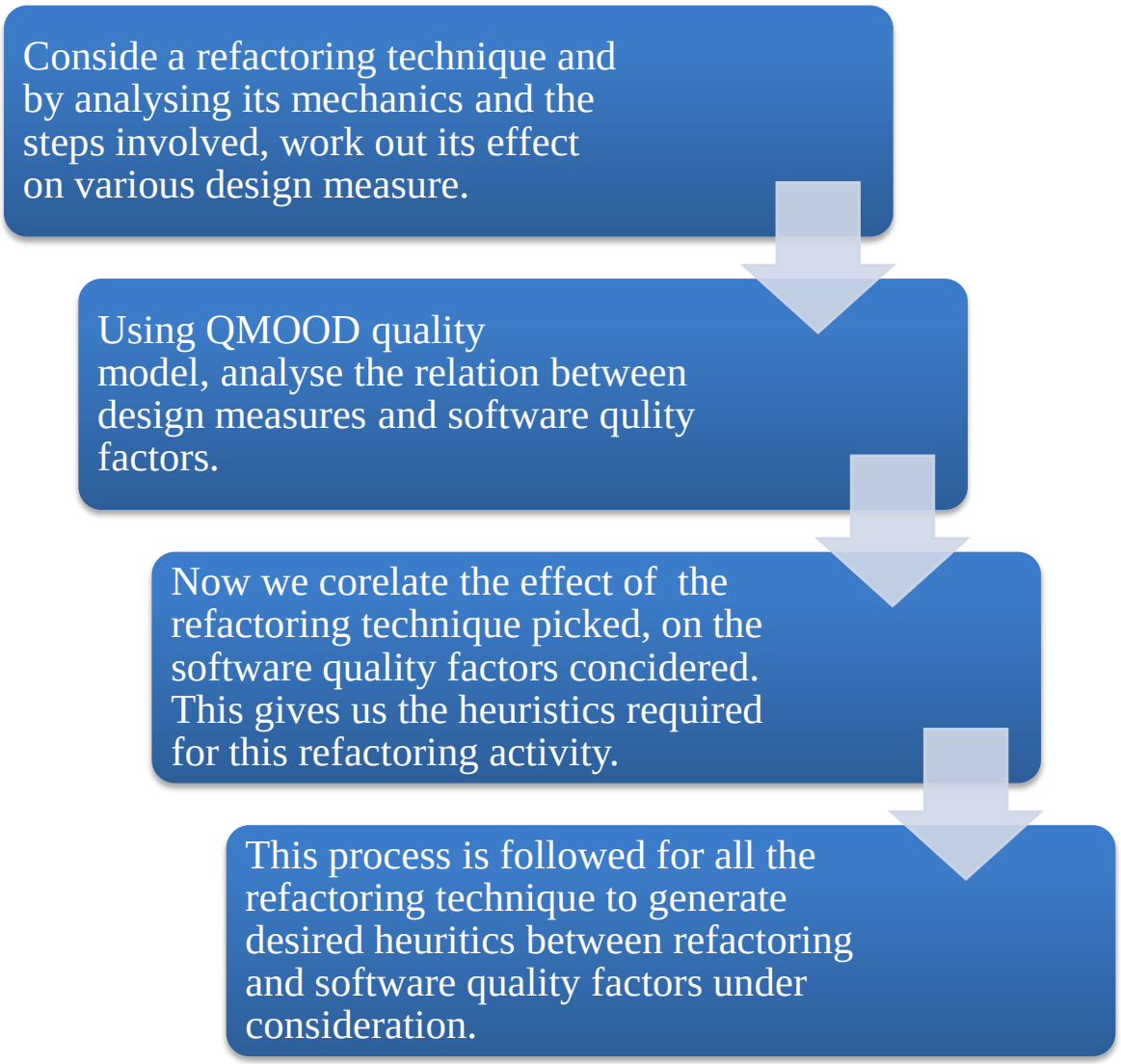


Figure 15: Flowchart for Research Methodology

4.2 Quality Model

A quality Model is basically a tool that helps us to define relation between the different design features of a software code and its quality factors. There are a number of models available that have been suggested by different gurus of software engineering. But here in this study the model that we have chosen is the hierarchical model suggested by Bansiya and Davis [2] for object- oriented quality assessment of software. It is a very comprehensive model and is referred to as QMOOD. This is a very refined model and has hierarchy of four different levels. These are mainly OO design components , OO design metrics, OO design properties and finally quality attributes. The earlier quality models suggested before QMOOD had a problem that they could be applied only after the product was complete. But QMOOD can be easily used for qualitative assessment of a software even in the early stages of its development. This is a huge advantages for any developer because every developer want to bring forth a perfect product and if he gets an assessment of the quality of the code at each step then he will definitely make required changes and put forward a product that completely conforms to the prerequisite standards, has no irregularities, in not un necessarily complex and has favorable properties. QMOOD is for sure a complete and comprehensive model that uses object oriented approach and gives us a definitive relationship between the internal design properties and the software quality. Bansiya and Davis [2], in their quality model have taken up six quality factors. In our study we have picked up four of them for analyses. We are going to develop heuristics between refactoring techniques and these software qualities. A detailed description of these qualities is as follows:

(1) Effectiveness: A code is said to have a high degree of effectiveness if it can attain a particular level of functionality and behavior when subjected to certain object oriented design concepts and techniques.

(2) Extendibility: This attribute refers to the existence and usage of properties in a system which allow it to be modified easily by incorporation of new features.

(3) Reusability: This is the property of a design allows it to be used or reapplied in a different problem with minimum effort and changes.

(4) Flexibility: Every design is made for a specific environment. Its flexibility refers to that characteristic of a design which allow it to be used in a different application by incorporation of some design changes. It shows the capability of the design to offer functionality after changes.

These above mentioned four quality factors are related to design properties of the software and this relationship is specified by QMOOD. What we are trying to do is that we know that we can use refactoring technique to change internal design properties of a software and these internal design properties are in turn related to the software quality factors (by QMOOD). Thus we can derive a heuristic between refactoring and quality attributes of a software. Each of the four quality factors mentioned above are a function of some design properties and are related as shown below:

Table 1: Relation between quality factors and design properties

QUALITY FACTORS	QUALITY INDEX CALCULATION
Reusability	–0.25 * Coupling + 0.25 * Cohesion + 0.5 * Messaging + 0.5 * Design Size
Flexibility	0.25 * Encapsulation – 0.25 * Coupling + 0.5 * Composition + 0.5 * Polymorphism
Extendibility	0.5 * Abstraction – 0.5 * Coupling + 0.5 * Inheritance + 0.5 * Polymorphism
Effectiveness	0.2 * Abstraction + 0.2 * Encapsulation + 0.2 * Composition + 0.2 * Inheritance + 0.2 * Polymorphism

As we can see, each software quality is actually a function of various software properties. Refactoring changes internal design features thus affecting the external quality factors too.

These results have been empirically validated by Bansiya and Davis [2], who proposed QMOOD model in the year 2002. To correctly indicate the effect of design properties on the quality attributes they initially started with weights +.5 or +1 for a positive influence and for a negative influence -0.5 or -1 were used. For the computed value of the software quality factor a range was fixed and it was 0 to $\pm$1. to keep the computed value within this range, the initial weighted values of design property influence on a quality factor were accordingly changed so that the total of the weighted values came out to be $\pm$1. This scheme was simple and its application was straightforward.

4.3 Design Metrics

In this section we define various software measures that are used to quantify system design properties. Refactoring changes these metrics. Am open source quality assurance tool ckjm is used to generate and collect the following metrices:

1. Design Size (DSC): It is a count of the number of classes in a particular system. Extract Class will raise the number of classes whereas Inline Class reduce this measure.
2. Number of Hierarchies (NOH): It refers to a tally of the number of class hierarchies in the design.
3. Average number of ancestors (ANA): This value signifies the averaged out value of the depth of inheritance tree (DIT).
4. Data access Metric (DAM): It is the ratio of the number of private (protected) attributes to the total number of attributes declared in the class.
5. Direct Class Coupling (DCC): It is a count of the different number o f classes that are class is directly related to. The metric includes classes that are directly related by message passing in methods and attribute declarations.
6. Cohesion among methods of Class (CAM): This computes the relatedness among methods of a class based upon the parameter list of methods. It is computed by summing up the intersection of parameters of a method with the maximum autonomous set of all parameter types in the class. A measure value close to 1.0 is preferred. (range 0 to 1)

7. Measure of aggregation (MOA): It's the measure of the extent of the part - whole relationship, realized by using attributes. It is a count of the number of data declarations whose types are user defined classes.
8. Measure of functional Abstraction (MFA): It is the ratio of the number of methods inherited by a class to the total number of methods that can be accessed by member methods of a class.
9. Number of Polymorphic methods (NOP): It is a count of the methods that can show polymorphic behavior. Such functions in C++ are marked as virtual.
10. Class Interface Size (CIS): It is a count of public functions in a class.
11. Number of Methods (NOM): It is a count of all the functions defined in a class.

4.4 Software Refactoring

In a previous section we discussed about the various refactoring techniques that Fowler [1] has suggested. In Fowler's Catalogue [1] we can see that each refactoring technique has been explained in a set pattern and there are following characteristics mentioned about each technique:

- when should refactoring be done
- What will be the advantages of refactoring and what will be the costs incurred
- A detailed explanation of the refactoring process in simple small steps
- A related illustration

As mentioned earlier in the previous chapter, Fowler in his catalogue [1] has listed around 22 code smells and the refactoring techniques that are possible to amend them are seventy two. But here in our study we have considered around forty three refactoring techniques for

analyses. In our study here we are looking at developing a refactoring heuristics that can guide potential developers about refactoring. The intention is to create heuristics that are simple to understand and apply and would beget clear positive results. Taking the help of QMOOD we created a documentation of the effect of applying refactoring on design properties. What we need is a relation between the refactoring technique being applied on the software to the change in quality factor of the software. To achieve the same we first assess the consequences of applying a refactoring technique on the design properties of the software. For this we pick one refactoring technique and apply it to a code. On doing this, various design properties of the code change and this change in design properties is then used in QMOOD to get the change in or the effect on software quality attributes. Thus we can deduce that how, on application of one refactoring technique, a code has gone through the process of evolution. Performing this for all the 43 techniques we gather and document the data. And after analyzing this data we can put forward a heuristics that can guide developer in deciding which refactoring technique is best suited for the changes that need to be done in the quality attributes of a software. To explain this methodology we take up an example as follows:

Refactoring technique considered: Extract Class

Where do we use Extract Class? Every developer will agree with the fact that with time every Class grows beyond what it was originally meant for. Responsibilities, data or operations keep getting added to it and soon it becomes too complicated, too big and understanding it is no less than a nightmare. Here comes a need to split the class.

What will Extract Class Technique do? When we have a class that is actually performing the work of multiple classes then we segregate it. We " Create a new class and move the relevant fields and methods from old class into the new field" ,[1]

How does Extract Class work and how will this affect the design properties of the code? For answering this we take a look at the mechanics involved and evaluate each step. To begin with, it is decided how the class responsibilities will be divided. Thereafter a new class is generated which has the split off responsibilities. Then a link is created between old class and new class. And last but not the least, use Move Field and Move Method to transfer fields and then methods to the new class.

What is the effect of the above procedure on the design properties of the code? The Coupling parameter increases as the two class entities are linked. Thus we can say that DCC measure increases. The CAM parameter also sees a boost because of an increase in cohesion of both new and old class (the process of splitting caused by Moving Field and Moving Method is responsible for that). This refactoring technique clearly has no impact on the properties that are linked to inheritance measures and thus show no change. These properties are namely NOP,MFA and ANA. This analysis has been summed up in the table below. Here an up arrow (↑) means increase and a down arrow (↓) denotes decrease. A (—) sign shows no effect.

Table 2: The impact of Extract Class on design measures

DSC	ANA	DCC	DAM	CAM	MOA	MEA	NOP	CIS
↑	—	↑	—	↑	↑	—	—	—

From this table we see that refactoring method of Extract Class has changed four measures of the class. When we use these results in QMOOD then we get that three out of the four considered software quality factors are improved. These attributes are Effectiveness, Flexibility and Reusability whereas one of the factors deteriorate. Now we are in a position to propose a heuristics that give us the impact of applying refactoring technique, Extract class on the software quality attributes.

As is clear from the above example that we have explained that to achieve a definitive relationship between different refactoring techniques and the various software quality attributes of a software we have to follow a threefold process which is as follows:

Step 1: Pick refactoring techniques from Fowler's catalogue [1], one at a time and review each small step of it.

Step 2: Enumerate every change that every small step of refactoring brings about in different design properties of the software.

Step 3: Use the QMOOD relations to then define the impact of these changed design measures on the quality factors of the software.

This will give us the heuristics that we are looking for. These heuristics will guide any software development process.

CHAPTER 5

ANALYSIS OF REFACTORING

This chapter primarily deals with assessment of the effect that various refactoring techniques have on different internal design properties of a software. Then we move on further to find out that how are the quality attributes like Effectiveness, Flexibility, Reusability and Extendibility are changed due to these refactoring activities. We are using QMOOD model to do the same. This model was initially suggested by Bansiya and Davis [2].

5.1 Establishing Refactoring Heuristics

In this section we pick up some of the Refactoring Techniques given by Fowler [1] and we evaluate their effect on various software metrics. Then we make two lists of Refactoring Techniques, first list tells us about those refactoring techniques that boost the software quality and the second list tells us about those refactoring techniques which make the software quality deteriorate. Developers can then use these heuristics to improve their software quality effectively in minimum time and effort.

As already discussed earlier, Fowler had organized various Refactoring Techniques under six heads. We study here, the effect of various techniques under each category in following six table. Each table shows us the effect of a particular technique on the various design metrics of software. Table 3 to Table 8 show us the same (Here an up arrow (↑) means increase and a down arrow (↓) denotes decrease. A (—) sign shows no effect). Also we applied these

refactoring techniques on a mobile user interface java code and derived the change in design metrics for it which were observed after refactoring. these values are also mentioned in the tables below :

Table 3: Effect of Composing methods

Refactoring	**DSC**	**ANA**	**DAM**	**DCC**	**CAM**	**MOA**	**MFA**	**NOP**	**CIS**
Replace Method with Method Object	↑(8)	—	—	↑(5)	—	↑(8)	—	—	—
Replace Temp with Query	—	—	—	—	—	—	—	—	↑(4)
Inline Method	—	—	—	↓(4)	—	—	—	—	↓(8)
Extract Method	—	—	—	↑(4)	↓(.7)	—	—	—	↑(8)
Substitute Algorithm	—	—	—	—	—	—	—	—	—

Table 4: Effect of moving features between objects

Refactoring	**DSC**	**ANA**	**DAM**	**DCC**	**CAM**	**MOA**	**MFA**	**NOP**	**CIS**
Introduce Foreign Method	—	—	—	↑(5)	—	—	—	—	↑(4)
Extract Class	↑(11)	—	—	↑(7)	↑(.8)	—	—	—	—

Introduce Local Extension	↑(8)	↑(5)	—	—	—	—	↑(.7)	↑(4)	↑(4)
Move Method	—	—	—	↓(3)	↑(.7)	—	—	—	↓(2)
Remove Middle	—	—	—	↑(8)	—	—	—	—	—
Inline Class	↓(11)	—	—	↓(7)	↓(.5)	—	—	—	—
Hide Delegate	—	—	—	↓(3)	—	—	—	—	—
Move Field	—	—	—	—	↑(.7)	—	—	—	—

Table 5: The effect of Organizing Data

Refactoring	**DSC**	**ANA**	**DAM**	**DCC**	**CAM**	**MOA**	**MFA**	**NOP**	**CIS**
Replace Type Code with Sub classes	↑(11)	↑(9)	—	—	—	—	↑(.5)	↑(7)	↑(6)
Replace array with object	↑(4)	—	↑(.7)	↑(5)	↑(.8)	↑(3)	—	—	↑(5)
Encapsulate Collection	—	—	—	↑(3)	—	—	—	—	↑(7)
Duplicate Observed Data	↑(5)	↑(6)	—	↑(7)	—	↑(5)	↑(.4)	↑(7)	↑(6)
Encapsulate Field	—	—	↑(.6)	—	—	—	—	—	↑(6)
Replace Type Code with Class	↑(7)	—	—	↑(5)	—	↑(5)	—	—	↑(5)

Change Bidirectional association to unidirectional	—	—	—	↓(4)	—	—	—	—	—
Replace Type Code with State/ Strategy	↑(3)	↑(4)	—	↑(3)	—	↑(4)	↑(.6)	↑(2)	↑(4)
Replace data value with object	↑(5)	—	—	↑(4)	—	↑(5)	—	—	↑(6)

Table 6 : The effect of Simplifying Conditional Expressions

Refactoring	**DSC**	**ANA**	**DAM**	**DCC**	**CAM**	**MOA**	**MFA**	**NOP**	**CIS**
Introduce Assertion	—	—	—	—	—	—	—	—	—
Decompose Conditional	—	—	—	—	—	—	—	—	↑(7)
Introduce Null Object	↑(9)	↑	—	—	—	—	↑(.5)	↑(6)	↑(7)
Replace conditional with Polymorphism	↑(3)	↑(4)	—	—	—	—	↑(.7)	↑(4)	↑(7)

Table 7: The effect of Making Method Calls Simpler

Refactoring	**DSC**	**ANA**	**DAM**	**DCC**	**CAM**	**MOA**	**MFA**	**NOP**	**CIS**
Introduce Parameter Object	↑(7)	—	—	↑(5)	—	—	—	—	—
Remove Setting Method	—	—	—	↓(4)	—	—	—	—	↓(3)
Remove parameter with Explicit Methods	—	—	—	—	—	—	—	—	↑(7)
Rename Method	—	—	—	—	—	—	—	—	—
Replace Parameter with Method	—	—	—	—	—	—	—	—	↑(7)
Preserve Whole Object	—	—	—	—	—	—	—	—	—
Remove Parameter	—	—	—	—	—	—	—	—	—

Table 8: The effect of Dealing with Generalizations

Refactoring	**DSC**	**ANA**	**DAM**	**DCC**	**CAM**	**MOA**	**MFA**	**NOP**	**CIS**
Extract Interface	↑(7)	↑(4)	—	—	—	—	↑(.7)	↑(4)	—
Pull Down Field	—	—	—	—	↑(.8)	—	—	—	—
Extract Sub Class	↑(9)	↑(4)	—	—	—	—	↑(.6)	↑(7)	↑(5)
Extract Super Class	↑(7)	↑(4)	—	—	—	—	↑(.7)	↑(6)	↓(4)
Replace delegation with Inheritance	—	↑(6)	—	↓	—	↓	↑(.7)	↑(8)	↓(5)
Collapse Hierarchy	↓(7)	↓	—	—	—	—	↓(.4)	↓(7)	—
Pull Up Method	—	—	—	↓(4)	—	—	↓(.6)	↓(4)	↓(7)
Replace Inheritance with Delegation	—	↓(6)	—	↑(5)	—	↑	↓(.4)	↓(5)	↑(4)
Pull Down Method	—	—	—	↑(4)	—	—	↑(.6)	↑(4)	↑(4)
Form Template Method	—	—	—	—	—	—	↑(.5)	↑(6)	↑(3)

This data presented here in form of tables gives us a clear idea that when any particular code is subjected to different refactoring techniques then its software quality measures get affected most of the time (not always, as a zero shows no impact). The analysis given also clearly

indicates that some of the software measures are affected by almost all the techniques while others are affected by only a few selected techniques. Also we see that there are some software measures that do not change for any of the considered refactoring techniques (although they can be changed by other refactoring techniques which have not been considered here in this study).

To draw a conclusive result from the above analysis we categorize all the software measures into two groups:

- The first group consists of those measures that are affected more by of the refactoring techniques considered here. We take up one category of refactoring techniques out of the six categorizes being considered here in this study. Then we deduce how many of its included techniques affect a particular measure. If that measure is found out to be impacted by more than half of the techniques then we consider that software measure to be highly co related to that particular category of techniques. We carry out the same for each category for that particular measure. Then we move on to the next measure and analyze the effect of various groups on it.
- The second group is the group of those measures that are affected by less than half of the techniques that are grouped under one category of refracting techniques. If this is the case then that particular measure is said to be loosely co related to that category of refactoring techniques.

The following table sums up all of this and gives us a heuristics that relates the effect of refactoring techniques to different software measure.

Table 9: Categorization of measures depending upon effect of refactoring on them

Refactoring Category	**High Impact**	**Low Impact**
(1) Making Method Calls Simpler	CIS	DCC,DSC
(2) Composing Methods	DCC & CIS	CAM, DSC & MOA
(3) Organizing Data	DCC,CIS,DSC& MOA	DAM,ANA MFA&NOP
(4) Moving Features Between Objects	CAM, DCC	CIS,DSC & MOA
(5)Simplifying conditional expressions	CIS	NOP,MFA,DSC&ANA
(6) Dealing with Generalization	MFA,CIS,NOP& ANA	DSC,CAM, DCC&MOA

From the above table we can very easily come to certain conclusion regarding some specific measures or some specific category of Refracting Techniques out of the six categories being considered over here. Some examples of such derivable conclusion from the table above are as follows:

- ✓ We can see that Class Interface Size or CIS is one measure that is impacted more than 50% of the times in a positive manner or it has a high impact for five of the categories.
- ✓ We see that if we need to perform refactoring to a code to have a positive impact on its Coupling (DCC) then it will be advisable to pick a technique from either Organizing Data or moving features between object or composing methods to get a positive result.

- ✓ Also we can derive after inspecting the second and the third column of the above table that if we need to change the Abstraction measure of the code then the most suitable category of the refraction techniques would be Dealing with Generalization.

Thus developers can use this analysis to pick up the most suitable refactoring technique whenever they have their focus on changing a particular software metric without any hit and trial. Also they can refrain from using any technique that will affect their code metrics negatively.

5.2 Correlation between Refactoring Techniques and Quality Factors

Now we move towards finding out how the various refactoring techniques change the software quality factors. We utilize the various refactoring heuristics that we found in the previous section. Quality factors can get either improved or can get deteriorated by various refactoring techniques. We have already found out impact of various refactoring techniques on software measures. Now using that we can calculate the effect of these refactoring techniques on software quality factors. Here is an example: If we consider Encapsulate Field we get three improvements and no deterioration therefore we can say that it is a safe technique which improves reusability, flexibility and effectiveness. Thus we get a correlation between the refactoring technique and its effect on the various quality factors of the software. Using the same kind of calculation we can categorize the various refactoring techniques into three categories:

(1) The techniques those are safe. They show more improvements than deteriorations.

(2) The techniques those are unsafe. They show fewer improvements and more deterioration.

(3) The techniques that do not have any effect on the software measures.

Now we have two tables that shows us this analysis:-

Table 10: Safe Refactoring techniques (Their effect on quality factors)

Refactoring	**Reusability**	**Flexibility**	**Extendibility**	**Effectiveness**	**Deterioration**	**Improvement**
Push Down Method	↑	↑	↑	↑	—	4
Introduce Null Objects	↑	↑	↑	↑	—	4
Replace Type Code With	↑	↑	↑	↑	—	4
Extract Sub Class	↑	↑	↑	↑	—	4
Duplicate Observed Data	↑	↑	↑	↑	—	4
Replace conditional with	↑	↑	↑	↑	—	4
Extract Interface	↑	↑	↑	↑	—	4
Form Template Method	↑	↑	↑	↑	—	4
Introduce local Extension	↑	↑	↑	↑	—	4
Encapsulate Field	↑	↑	—	↑	—	3
Change bidirectional association to unidirectional	↑	↑	↑	—	—	3
Hide delegate	↑	↑	↑	—	—	3
Extract Super Class	—	↑	↑	↑	—	3

Replace Array with Method	↑	↑	↓	↑	1	3
Replace Method with Object	↑	↑	↓	↑	1	3
Replace Data Value with Object	↑	↑	↓	↑	1	3
Extract Class	↑	↑	↓	↑	1	3
Replace Delegation With Inheritance	↓	↑	↑	↑	1	3
Replace Type Code With Class	↑	↑	↓	↑	1	3
Move Method	—	↑	↑	—	—	2
Remove Setting Method	↓	↑	↑	—	1	2
Inline Method	↓	↑	↑	—	1	2
Replace Parameter with Explicit Methods	↑	—	—	—	—	1
Decompose Conditional	↑	—	—	—	—	1
Move Field	↑	—	—	—	—	1
Replace parameter with Method	↑	—	—	—	—	1
Replace Temp with query	↑	—	—	—	—	1
Push Down Field	↑	—	—	—	—	1

Table 11: Unsafe Refactoring Techniques and their effect on Quality Factors

Refactoring	**Reusability**	**Flexibility**	**Extendibility**	**Effectiveness**	**Deterioration**	**Improvement**
Pull Up Method	↓	↓	↓	↓	4	0
Collapse Hierarchy	↓	↓	↓	↓	4	0
Remove Middle Man	↓	↓	↓	—	3	0
Replace Inheritance with Delegation	↑	↓	↓	↓	3	1
Inline Class	↓	↓	↑	↓	3	1
Extract Method	—	↓	↓	—	2	0
Encapsulate Collection	↑	↓	↓	—	2	1
Introduce Foreign Method	↑	↓	↓	—	2	1
Introduce Parameter Object	↑	↓	↓	—	2	1

The refactoring Techniques which do not affect the Software Quality factors are as follows:

- Preserve Whole Object
- Rename Method
- Introduce Assertion
- Remove Parameter
- Substitute Algorithm

These heuristic results that have been given in the above two tables can be used by developers to achieve improvement in quality factors as desired. It can be seen that it is now easier to decide which refactoring techniques should be used to effectively improve software qualities and which refactoring techniques should be used with precautions as they can have a negative impact on various quality factors of the software. For example, if the Reusability of the system is to be improved then the developer needs to use all those techniques that have been shown to increase reusability. We can clearly see this now that if we need a refactoring technique that would, let us say, improve its reusability, would make it much more flexible as far as changing classes goes and would also make extending its class a less tedious task then it would be good idea to select the activity of Hide Delegate. It suits our requirements. There is now no need to go for other refactoring techniques which might potentially harm our code instead of improving it. These results will help in selecting the most optimum refracting technique which will reduce a lot of effort of the developer. Now the essential design changes can be brought about efficiently in minimum time, minimum risk and probably less costing. We can also pick up a particular quality factor and categorize refactoring techniques as safe or unsafe for it. we just have to reorganize the above tables for it.

5.3 Impact analysis of refactoring on quality factors in statistical form

In our study we considered four quality factors to study impact of refactoring techniques on them, whether they improve or decrease or if they had no effect at all when subjected to techniques. These four qualities were: Reusability, Flexibility, Extendibility and Effectiveness. What we are trying to project here is that what proportion or percentage of total refactoring techniques being used by us actually improve or harm the four quality factors under consideration and what is the tentative percentage of the total refactoring activities performed in this study that do not change the factors at all. So accordingly we have three categories of the refactoring techniques as unchanged, deteriorate and Improved. This is shown in a table below. The details of the same are as follows:

Table 12: Statistics relating the effect of refactorings to quality factors

Quality Factor	**Improved (in%)**	**Unchanged(in%)**	**Deteriorate(in%)**
Reusability	65	19	16
Flexibility	53.5	25.5	21
Extendibility	42	28	30
Effectiveness	42	49	9

This shows that Reusability improves for 65 % of the refactoring techniques that have been considered here and that only 9 % of those activities have been found to have a negative impact on the reusability of the software. This gives a pretty fair idea to the developer that the refactoring is more probable to make the software more usable. If we considered Effectiveness then it improves by 42% of the refactoring techniques and only 9% of the techniques considered make it deteriorate. We see that mostly there is a larger proportion for

improvement of the quality that deterioration. This is sure to guide the developers through refactoring for software quality improvement.

5.4 Validation

We validated our study by conducting a survey with 30 developers who are working in real world industrial environment. They were asked to fill up a survey form which gave details about the effect that specific refactoring techniques had on their code quality. They took suggestions from the heuristics given by this study for choosing a particular technique to have desired effect on a software quality. Then after refactoring they were asked to take up the given 15 questions and consider them on a scale of 0 to 5 where 0 meant not satisfactory at all and 5 meant perfect achievement of desired result on refactoring [Appendix A & Appendix B].

The data collected clearly indicated that the suggested heuristics was very helpful in selecting the best suited refactoring technique and achieving desired results.

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 Conclusion

Every developer is faced with the need to improve the quality of his software at some point of time. This is essential to keep the code easily maintainable and to avoid the decay of its structure with constant changes which need to be done with changing requirements. This can be effectively done by using refactoring techniques . Fowler has provided the developers with a catalogue of refactoring techniques [1] which can be used by them.

The results of this study provide the developers with an insight on the relationship between the impact of different refactoring techniques on the quality factors of the software. Not all refactoring techniques improve the quality factors of the software. It has been shown that some refactoring techniques improve the quality factors while other refactoring techniques might have a negative impact on software quality. Using the heuristics presented in of this study, a developer can follow a goal oriented refactoring process which can improve Reusability, Flexibility, Extendibility and Effectiveness of his software. Also he can avoid using the technique which will deteriorate a specific quality factor. For example if the developer need to improve the reusability of a software then he must use those refactoring techniques that have been found to improve this particular quality factor and avoid using

those refactoring techniques which deteriorate the reusability of the software. Thus these heuristics will help developers to use refactoring techniques more efficiently and objectively.

6.2 Limitations and Future work

This study presented here has its own limitations. These limitations can be summed up as follows:

- In our study we have considered some of the refactoring techniques suggested by Fowler [1]. But there are a lot of variations and hybrid forms of those refactorings possible. It is possible that the results might vary when used with a different pool of refactoring techniques.
- Initially while describing evolution of the code we started from the point where we suggested that Refactoring is wholly responsible for evolution process of any code. But it is not so. Codes evolve with time in the actual environments because of various other reasons too like language or library evolution and work around techniques applied to make code rejuvenate. Many process are done to improve the behavior of the code in a different environment than its original one. Such factors too evolve the code.
- The set of quality metrics used by us is limited. This needs to be extended to many other quality factors that are linked to all the quality factors related to a code.
- The current study has been performed on a particular type of mobile software code and this can be conducted on a number of variety of code / software. In this way we can infer that which refactoring technique is more relevant for which type of software code.

It will be a humble effort in all our future work to attract the attention of the entire scientific community towards the fact that refactoring can actually have a very positive impact in a very effective way on the code quality if it is guided by similar heuristics which have been validated through a formalized model. In future we intend to take into consideration following points:

- To do the same study for a larger pool of software quality factors. Here we have considered the effect of refactoring on four factors only. We would like to make the list exhaustive so as to cover all the factors which decide the quality of a code.
- To generate heuristics for other refactoring techniques. There are a number of refactoring techniques which can be used for software quality improvement and it will be our effort to draw heuristics for as many as possible.

REFRENCES

[1]. M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts, Refactoring: Improving the Design of Existing Code, Addison Wesley, 1999.

[2]. J. Bansiya, C. Davis, ―A Hierarchical Model for Object-Oriented Design Quality Assessment‖, IEEE Transactions on Software Engineering, 28 (1), (2002) pp. 4–17.

[3]. B. Meyer, Object-Oriented Software Construction, Prentice Hall, second ed., 1997.

[4]. T. Mens, S. Demeyer, B.D. Bois, H. Stenten, P. Van Gorp, ―Refactoring: Current Research And Future Trends‖, Electronic Notes in Theoretical Computer Science, 82 (3), (2003) pp. 483–499.

[5]. 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.

[6]. 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.

[7]. 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.

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

[9]. B.D. Bois, S. Demeyer, J. Verelst, ―Refactoring–Improving Coupling and Cohesion Of Existing Code‖, In Belgian Symposium on Software Restructuring, Gent, Belgium, pp. 144–151, 2005.

[10]. J. Ratzinger, M. Fischer, H. Gall, ―Improving Evolvability Through Refactoring‖, Proceedings of the 2nd International Workshop on Mining Software Repositories (MSR'05),1–5.

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

[12] www.refactoring.com/catalog

[13] L. Tahvildari, ―Quality-Driven Object-Oriented Re-engineering Framework‖. PhD Thesis. Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, Ontario, Canada, 2003.

[14] 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.

[15] www.martinfowler.com/books/refactoring.html

[16] L. Tahvildari, and K.A. Kontogiannis, "Metric-Based Approach to Enhance Design Quality through Meta-Pattern Transformations", In Proc. European Conf. Software Maintenance and Reeng., 2003, pp. 183-19

[17] Y. Yu, J. Mylopoulos, E. Yu, J.C. Leite, L. Liu, E.H. D'Hollander, "Software refactoring guided by multiple soft-goals", In Proceedings of the 1st workshop on Refactoring: Achievements, Challenges, and Effects, in conjunction with the 10th WCRE conference 2003, Victoria, Canada, November 13-16, 2003, pp. 7-11

[18] T. Mens, and T.A. Tourwé, "Survey of Software Refactoring", IEEE Transactions on Software Engineering, 30(2): 126-139,February, 2004.

[19] S. Demeyer, S. Ducasse, O. Nierstrasz, "Finding Refactorings via Change Metrics", In Proceedings of the 15th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA'00, Minneapolis, USA, 2000.

[20] D. Coleman, D. Ash, B. Lowther, and P. Oman. Using metrics to evaluate software system maintainability. IEEE Computer, pages 44–49, August 1994.

[21] M. Alshayeb, ―Empirical investigation of refactoring effect on software quality‖, Information and Software Technology, vol. 51, pp.1319–1326,2009

[22] D. Dig, C. Comertoglu, D. Marinov, R. Johnson, ―Automated Detection of Refactorings in Evolving Components‖, Proceedings of European Conference on Object-Oriented Programming (ECOOP'06), pp. 404–428.

[23] N. Hsueh, P. Chu, W. Chu, ―A Quantitative Approach for Evaluating the Quality of Design Patterns‖, The Journal of Systems and Software, 81(8), (2008) 1430–1439.

[24] J. Bøegh, S. Depanfilis, B. Kitchenham, A. Pasquini, A Method for Software Quality Planning, Control, and Evaluation, IEEE Software, Researcher's Corner, 16 (2), (1999) pp. 69–77.

[25] F. Dandashi, D.C. Rine, ―A Method for Assessing the Reusability of Object-Oriented Code Using A Validated Set of Automated Measurements‖, Proceedings of 17th ACM Symposium on Applied Computing, pp. 997–1003, 2002.

[26] N. Hsueh, P. Chu, W. Chu, ―A Quantitative Approach for Evaluating the Quality of Design Patterns‖, The Journal of Systems and Software, 81(8), (2008) 1430–1439.

[27] B. Henderson-Sellers, Object-Oriented Metrics: Measures Of Complexity. Prentice-Hall, 1996.

[28] V. Basili, L. Briand, and W. Melo, "A Validation of Object-Oriented Design Metrics as Quality Indicators," IEEE Transactions on Software Engineering, vol. 22, pp. 751-761,

Oct. 1996 1996.

[29] L. Briand, J. Wust, J. Daly, and V. Porter, "Exploring the relationships between design measures and software quality in object-oriented systems," The Journal of Systems and

Software, vol. 51, pp. 245-273, 2000 2000.

[30] K. Stroggylos and D. Spinellis, "Refactoring - Does it improve software quality?," in 5th International Workshop on Software Quality (WoSQ'07: ICSE Workshops), 2007, pp. 10-16.

[31] K. Elish and M. Alshayeb, "A Classification of Refactoring Methods Based on Software Quality Attributes," The Arabian Journal for Science and Engineering, vol. 36, 2011

[32] . Tahvildari, K. Kontogiannis, and J. Mylopoulos. Quality-driven software re-engineering.

Journal of Systems and Soft-ware, 66(3):225–239, June 2003

APPENDIX A

The questionnaire used for the industrial survey which was done to validate the results:

Q1: Has the execution of the functions become clearer?

Q2: Are the results being generated by the software as per the expectations?

Q3: Has it become easier to control the software?

Q4: Whenever there is a failure in the system has it become easier to detect its source?

Q5: Has it become easy to adapt this software into different environments?

Q6: Has there been any improvement in the response time of this software?

Q7: Have the future aspects of the evolving the code become better?

Q8: Does the software show any enhanced capacities for processing by multi-users?

Q9: When a failure occurs what is the ease of recovering the data which has been lost?

Q10: Has Data entry by user become easier?

Q11: Has it become more convenient to change the software mission?

Q12: If any particular component of the software needs a change, is it convenient to do so?

Q13: Has there been an improvement is anomaly management?

Q14: Is there any improvement in the capability of the software to prevent any defects?

Q15: Is it now easier to expand the software?

APPENDIX B

The data collected from the industrial survey that validates the result:

SNo.	Question	Ratings given		
		Low (<=2)	Average (=3)	High (>3 & <=5)
1.	Has the execution of the functions become clearer?	1	6	23
2.	Are the results being generated by the software as per the expectations?	4	5	21
3.	Has it become easier to control the software?	2	8	24
4.	Whenever there is a failure in the system has it become easier to detect its source?	5	8	17
5.	Has it become easy to adapt this software into different environments?	2	9	19
6.	Has there been any improvement in the readability of this software?	2	2	26
7.	Have the future aspects of the evolving the code become better?	1	4	25

8.	Does the software show any enhanced capacities for processing by multi-users?	3	9	18
9.	When a failure occurs what is the ease of recovering the data which has been lost?	2	9	19
10.	Has Data entry by user become easier?	0	10	21
11.	Has it become more convenient to change the software mission?	4	7	19
12.	If any particular component of the software needs a change, is it convenient to do so?	4	6	20
13.	Has there been an improvement is anomaly management?	0	11	19
14.	Is there any improvement in the capability of the software to prevent any defects?	3	5	22
15.	Is it now easier to expand the software?	2	4	24
	Mean Value	2.3 (7 %)	6.8 (23%)	21.1 (70 %)