An Empirical Exploration of Refactoring effect on Software Quality using External Quality Factors

S.H. Kannangara and W.M.J.I. Wijayanayake

***Abstract*— Quality software are robust, reliable and easy to maintain, and therefore reduces the cost of software maintenance. Since software systems undergo modifications, improvements and enhancements to cope with evolving requirements, quality of software can be decreased. While software system is evolving, refactoring is one of the methods which have been applied with the purpose of improving the software quality. Refactoring is defined as the process of improving the design of the existing code by changing its internal structure without affecting its external behavior, with the main aims of improving the quality of software product. Therefore, there is a belief that refactoring improves quality factors such as understandability, flexibility, and reusability. However, there is limited empirical evidence to support such assumptions.**

The objective of this study is to validate/invalidate the claims that refactoring improves software quality. Experimental research approach was used to achieve the objective and ten selected refactoring techniques were used for the analysis. The overall impact of selected refactoring techniques and the impact of individual refactoring technique were assessed based on external measures namely; analyzability, changeability, time behavior and resource utilization.

After analyzing the experimental results, overall analysis ended up with the result that refactoring deteriorates the code quality. However, individual analysis shows that some refactoring techniques improve the code quality while rest is deteriorating the code quality. Furthermore, among the tested ten refactoring techniques, "Replace Conditional with Polymorphism" ranked in the highest as having high percentage of improvement in code quality and "Introduce Null Object" was ranked as worst which is having highest percentage of deteriorate of code quality among the analyzed ten refactoring techniques.

***Index Terms*— Refactoring, Software Maintenance, Code Quality Improvement, Code Quality Measures, ISO 9126**

I. INTRODUCTION

Software quality can be described as the conformance to functional and non-functional requirements, which are related to characteristics described in the ISO-9126 standard namely reliability, usability, efficiency, maintainability and portability [1]. The factors that affect software quality can be classified into two groups [2]: factors that can be directly measured i.e. internal quality attributes (e.g. Coupling, Cohesion, Line of Code and etc.) and factors that can be measured only indirectly i.e. external quality attributes (e.g. understandability, analyzability and etc.).

Quality software are robust, reliable and easy to maintain, and therefore reduces the cost of software maintenance [3]. Therefore, developers and designers always strive for quality software. However, any useful software system requires constant evolution and change. While software system is evolving, maintaining the software quality is one of the vital factors in software maintenance process.

As the software system is enhanced, modified and adapted to new requirements, the code become more complex and drifts away from its original design. Since, the major part of total software development cost is devoted to software maintenance. Maintenance of software is reported as a serious cost factor [4] and as stated in [5], over 90% of the software development cost is for software maintenance.

Software maintenance best practices are arising with the purpose of a better evolution of software while preserving the quality of software systems. One solution proposed to reduce the software maintenance effort is software code refactoring [6] which is a method to continuous restructure code according to implicit micro design rules. According to the Fowler's definition [6], refactoring is the change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior. Refactoring is by definition supposed to improve the maintainability of a software product; however, its effect on other quality aspects is unclear. Therefore, there are hot and controversial issues about refactoring.

As stated by Mens and Tourwé [4], refactoring is assumed to positively affect non-functional aspects, likely extensibility, modularity, reusability, complexity, maintainability, and efficiency. Recently Bois and Mens [7] performed a return on investment analysis on an open source project, in order to estimate savings in effort, given a specific code change. They found that, most of the time, refactoring has beneficial impacts on maintenance activities, and thus are motivated from an economical perspective. However, additional negative aspects of refactoring are reported, too [4]. They consist of additional

Manuscript updated on February 12, 2014. Recommended by Dr. Damitha Karunarathne on July 10,2014.

S.H. Kannangara is with the Department of Industrial Management, Faculty of Science, University of Kelaniya, Sri Lanka. (e-mail: Sandeepa.kannangara@gmail.com).

W.M.J.I. Wijayanayake is also with the Department of Industrial Management, Faculty of Science, University of Kelaniya, Sri Lanka. (e-mail: janaka@kln.ac.lk).

memory consumption, higher power consumption, longer execution time, and lower suitability for safety critical applications.

Several studied have been conducted to evaluate the impact of refactoring of software quality ([8], [9]). Even though some of those studies claim that refactoring improves the quality of software, most of them did not provide any quantitative evidence. Therefore, the empirical evidence of the effect of refactoring is rarely to be found [10]. Moreover, there is lack of studies which identified the most beneficial refactoring techniques among available large number of refactoring techniques. As mentioned by Stroggylos and Spinellis [11], 'effect of a refactoring on the software quality' is a one of the open issues that remain to be solved.

Altogether, the real advantages and disadvantages of refactoring are still to be fully assessed. As regards quality, it appears to be a convergence of positive remarks, still, without solid quantification. Furthermore, there are few quantitative evaluations of impact of each refactoring techniques to the software quality. It is sometimes difficult to judge whether the refactoring in question should be applied or not without knowing the effect accurately. Especially in software development industry, from the viewpoint of project managers, it is imperative to quantitatively evaluate the effect of refactoring on software quality before applying it. Without knowing which refactoring technique will be more beneficial in terms of quality, managers cannot judge whether they should go for refactoring or not because they have to be cost sensitive. Therefore, there is a need of study which can quantitatively evaluate the impact of each refactoring technique on quality of code.

The objective of this study is to evaluate the real effect of refactoring on code quality using external measures. Moreover, to identify the refactoring techniques which have highest positive impact on code quality that can help software developers to select most beneficial refactoring techniques.

The reminder of this paper structured as follows: Section 2 provides a summary of relevant literature which addressed the relationship between refactoring and software quality. Experimental design which is used for the research is described in Section 3. Section 4 provides experimental data analysis. Finally, the section 5 provides the discussion of results and section 6 provides the conclusions and suggestions for future research that can be pursued in this area.

II. Related Work

Studies which have been conducted to evaluate the impact of refactoring of software quality can be categorized into mainly three categories according to focused quality factors: internal quality factors, external quality factors and combination of both quality factors.

Even though some of those studies claim that refactoring improves the quality of software, most of them do not provide quantitative evidence. However, few researches quantitatively evaluated whether refactoring indeed improves quality (e.g. [8], [9]) and came up with different results.

Among them, significant number of studies quantitatively evaluated the impact of refactoring using internal quality attributes. Bois and Mens [7] proposed a technique using metrics to analyze the refactoring impact on internal quality metrics as indicators of quality factors. They proposed formalism based on abstract syntax tree representation of the source-code, extended with cross-references to describe the impact of refactoring on internal program quality. They focused on three refactoring methods: "Encapsulate Filed", "Pull up Method" and "Extract Method". However, they did not provide any experimental validation in an industrial environment. The results of their work showed both positive and negative impacts on the studied measures. Stroggylos and Spinellis [11] analyzed source code version control system logs of four popular open source software systems to detect changes marked as refactoring and examine their effects on software metrics. They finally came up with a conclusion that refactoring does not improve quality of a system in a measurable way. Bois et al. [12] developed practical guidelines for applying refactoring methods to improve coupling and cohesion characteristics and validated these guidelines on an open source software system. There were only five refactoring techniques under study: Extract Method, Move Method, Replace Method with Method Object, Replace Data Value with Object, and Extract Class. They assumed that coupling and cohesion are internal quality attributes which are generally recognized as indicators for software maintainability. At the end they came up with results that the effect of refactoring on coupling and cohesion measures ranged from negative to positive. Kannangara and Wijayanayake [13] evaluated both overall and individual impact of selected refactoring techniques. Ten refactoring techniques were evaluated by them through experiments and assessed five internal measures: Maintainability Index, Cyclomatic Complexity, Depth of Inheritance, Class Coupling and Lines of Code. They used source codes developed using C#.net and internal measures were extracted through Visual Studio IDE. According to their findings, only maintainability index indicated an improvement in code quality of refactored code than non-refactored code and other internal measures did not indicate any positive effect on refactored code.

Few other studies took the approach of assessing the refactoring effects on external software quality attributes. Geppert et al. [14] empirically investigated the impact of refactoring on changeability. This study found that the customer reported defect rates and change effort decreased in the post-refactoring releases. The effect of refactoring on maintainability and modifiability was investigated by Wilking et al. [9] through an empirical evaluation. Maintainability was tested by randomly inserted defects into the code and measuring the time needed to fix them. Modifiability was tested by adding new requirements and measuring the time and Line of Code (LOC) metric needed to implement them. Their findings on maintainability test show slight advantage for refactoring and Modifiability test shows disadvantage for refactoring. The impact of ten individual refactoring techniques

empirically evaluated by Kannangara and Wijayanayake [15, 16] using four external measures: Resource Utilization, Time Behavior, Changeability and Analyzability which are ISO sub External Quality factors. Their experimental results indicated that there are no quality improvements in refactored code for majority of the selected refactoring techniques.

Other remaining studies used the approach of assessing the impact of refactoring on internal attributes as indicators of external software attributes. To do so, they defined and relied on relationships between internal and external attributes defined by different authors (ex. [17]). Kataoka et al. [8] proposed coupling metrics as a quantitative evaluation method to measure the effect of refactoring on program maintainability. For the purpose of validation they analyzed a C++ program for two refactoring techniques: Extract Method and Extract Class which developed by a single developer, however did not provide any information on the development environment. Thus, it is questionable if their findings are valid in a different context where development teams follow a structured process and use common software engineering practices for knowledge sharing. Moser et al. [18] proposed a methodology to assess whether the refactoring improves reusability and promotes ad-hoc reuse in an Extreme Programming (XP)-like development environment. They focused on internal software metrics that are considered to be relevant to reusability based on metric interpretation of Dandashi and Rine's work [17]. They came up with a conclusion that refactoring has a positive effect on reusability. The impact of refactoring on development productivity and internal code quality attributes was analyzed by Moser et al. [19]. A case study has been conducted to assess the impact of refactoring in a close-to industrial environment and the collected measures were Effort (hour), and Productivity (LOC). Results indicate that refactoring not only increases aspects of software quality, but also improves productivity. Alshayeb [3] quantitatively assessed the effect of refactoring on different external quality attributes: Adaptability, Maintainability, Understandability, Reusability, and Testability using software matrices based on metric interpretation of [17]. However, this study didn't prove that refactoring improves external quality of the software. Shatnawi and Li [20] studied the effect of software refactoring on software quality. They have conducted the study on a larger number of refactoring techniques (43 refactoring) and measured four external quality factors indirectly using nine different internal software quality measures based on Quality Model for Object Oriented Design (QMOOD). They had provided details of findings as heuristics that can help software developers make more informed decisions about what refactoring techniques to perform in regard to improve a particular quality factor. They validated the proposed heuristics in an empirical setting on two open-source systems. They found that the majority of refactoring heuristics do improve quality; however some heuristics do not have a positive impact on all software quality factors.

After analyzing the above mentioned studies, several concerns in those can be deduced as follows:

- All these previous studies did not come up with same conclusions regarding the impact of refactoring. Therefore, there is further need of analyzing the impact of refactoring.
- Most of the studies which were evaluated external quality factors did it by using internal quality factors and majority of them used quality models. Therefore, their research findings are totally depending on the validity of those quality models.
- Those who evaluated external quality factors only focused one or two external quality factors. None of them focus on ISO quality factors or other world accepted quality model for the selecting quality factors.
- Except one study [20] all the other studies used only less than ten refactoring techniques for their evaluation. Most of them did not consider any valid justification when selecting refactoring techniques for their study.
- As most of the studies did not evaluate large number of refactoring techniques, they cannot be able to identify the most beneficial refactoring techniques among catalogue of large number of refactoring techniques.
- Finally, none of previous studies did the evaluation of impact of individual refactoring techniques and evaluation of overall impact of those refactoring techniques in the same study.

III. Experimental Design

Experiential evidence for the effect of refactoring is rarer to be found. Those experiments were ended up with mixed picture of refactoring. Therefore, experimental research approach is selected to quantitatively access the overall impact of all the selected refactoring and the impact of individual refactoring technique separately.

The general approach followed by experiment was consisting two groups. One group was assigned refactored code using selected refactoring technique or techniques while the rest was assigned non-refactored source code. The assignment to a treatment and control groups were done randomly.

A. Selected Refactoring Techniques

Fowler [6] proposed 72 refactoring techniques in his catalogue of refactoring. Among the studies which have evaluated the impact of refactoring, the most recent study [20] presented large evaluation of 43 refactoring techniques among 72 refactoring techniques in Fowler's [6] catalogue. In there, the evaluated refactoring techniques were ranked according to the impact of code quality. Therefore, for this study, ten refactoring techniques were selected from Shatnawi and Li's [20] study which were ranked as having a high impact.

Selected Refactoring Techniques are:

- Introduce Local Extension
- Duplicate Observed Data
- Replace Type Code with Subclasses
- Replace Type Code with State/Strategy
- Replace Conditional with Polymorphism
- Introduce Null Object
- Extract Subclass

- Extract Interface
- Form Template Method
- Push Down Method

B. Selection of Source Code

Refactoring is a technique which is mostly related with object oriented programming. Therefore, the selection of development environment and programming language was done mainly based on the above reason.

Java, C# and C++ are the some of the most popular object oriented programming languages which are being used in the current IT industry. Among those, Java and C++ are the commonly used programming languages in previous studies which evaluated the impact of refactoring on code quality improvement (e.g. [8], [20]).

Therefore, C# was selected as the programming language and Visual Studio as the development environment for this study.

To apply each refactoring technique separately, mini size applications were selected as source codes. Most of those codes were from mini scale game applications which are freely available on World Wide Web. One relevant bad smell was identified and one suitable refactoring technique was applied among selected 10 refactoring techniques to each selected source code. The average line of codes per each selected application was around 300. Finally the ten refactored source codes were available for the experiment with 10 original source codes of them.

In order to apply 10 refactoring techniques together small scale project with bad smells was selected as the source code. The selected application was a system which was developed in the Department of Industrial Management, University of Kelaniya for its academic staff to schedule their personal and professional events and to manage their online documents repository. The source code contained around 4500 lines of codes. The relevant bad smells were identified and all the selected refactoring techniques were applied to the source code.

C. Selected Quality Factors

As there are only few studies which evaluated the impact on refactoring on external quality factors without using internal quality factors, this experiment was designed to evaluate the external quality factors without using any internal quality factors or quality models.

As stated by Al-Qutaish in his study [21], ISO 9126-1 quality model is the most useful one, since it has been built based on an international consensus and agreement from all the country members of the ISO organization. Therefore, ISO quality model [1] is used for the selection of quality factors.

Following are the external quality attributes which are selected from ISO quality attributes for this study:

1. Maintainability: Maintainability is a set of attributes that bears the effort needed to make specified modifications. Following sub characteristics were tested in this study [1].
   i. Analyzability
   ii. Changeability
2. Efficiency: Efficiency is a set of attributes that bear on the relationship between the level of performance of the software and the number of resources used, under stated conditions. Following sub characteristic will be tested in this study [1].
   iii. Resource Utilization
   iv. Time Behavior

Other quality factors in ISO quality model have to be excluded from this study. The functionality factor was excluded, because refactoring does not change the behavior of systems, rather it changes the internal characteristics of the systems without changing functionality. Usability factor was excluded, because it is more end user oriented. Usability indicates how it is easy to learn and use software as an end use application, not about the source code. Reliability is implementation oriented quality factor. Reliability is an attribute that can only be estimated for live software applications with a variety of test data and then inspecting the defects uncovered or the number of times that the code terminates normally with the expected output. Therefore, reliability also excluded from the study. Portability indicates level of flexibility to migrate software to a different hardware or an Operating system. However, in this experimental design there is no direct way to evaluate this factor. Therefore, this factor has also been excluded from the study.

D. Variables and Measurements

1. Independent Variables:

The independent variable for this experiment is the treatment which is a single, dichotomous factor. Either a participant is assigned to a group which uses a refactored code or to a group which uses a code without refactoring, in order to rule out the placebo effect which known as a phenomenon which may result in some therapeutic effect if subjects are given control [22].

2. Dependent Variables:

The Dependent variables for this experiment are,

- Marks obtained for question papers
- Time need to fix bugs
- Task Execution Time
- Memory Consumption to execute task

E. Research Hypothesis

This study was aimed at presenting evidence that would allow rejecting (or accepting) the following four hypotheses:

- Analyzability

  H_0A: Analyzability of refactored code is lower than non-refactored code.

  H_1A: Analyzability of refactored code is higher than non-refactored code.
- Changeability

  H_0B: Changeability of refactored code is difficult than non-refactored code.

  H_1B:Changeability of refactored code is easier than non-refactored code.

- Time Behavior
  H_0C: Response time of refactored code is longer than non-refactored code.
  H_1C: Response time of refactored code is shorter than non-refactored code.
- Resource Utilization
  H_0D: Efficient utilization of computer Resources is low for refactored code than non-refactored code.
  H_1D: Efficient utilization of computer Resources is higher for refactored code than non-refactored code.

F. Sample Selection

The experiment was carried out with set of sixty students firstly to access the individual impact of refactoring techniques separately and set of twenty students secondly to access the overall impact of selected refactoring techniques. When selecting participants, the major skill that should have with participants was decided as a programming skill. Current undergraduates and recently passed out students of the University of Kelaniya were selected as the population for experimental sample selection.

The selection procedure was conducted for undergraduates and recently passed out students based on two criteria. They are,

- Based on semester examination results for programming related subjects
- Based on survey results done in order to identify student's familiarity of C#.Net and Object Oriented Concepts: Online questionnaire was designed to gather responses.

After collecting students' results and responses, those were aggregated and scaled to ten. Average values for each student was calculated and ranked them according to the average. Then the selection of students for the experiment was done according to their rank starting from top ranks.

Group size was decided as 3 members per one group for the first experiment or the analysis of each refactoring techniques separately. Due to availability of limited resources at Undergraduate laboratories and controlling of large groups is not possible with available human resources, the required number of participant for the second experiment was limited to 60.

For the second experiment or to analyze all the selected refactoring techniques together, group size was decided as 10 members per one group due to the same reason.

G. General Procedure

The general procedure for both experiments: analysis of overall impact of all the selected refactoring techniques and analysis of individual impact of refactoring techniques was mainly carried out in two steps. The first step of each experiment was done with controlled and experimental groups. The second step for each experiment was carried out in a software testing environment, in order to collect resource utilization and time behavior measures.

- Step 1:

The execution of the experiment started with an oral presentation by introducing application which is being used for the experiment, the experimental environment with procedure, and the general conditions of the experiment.

After that, an initial test was carried out in order to assess the impact on refactoring of code analyzability. Initially several minutes were provided to both groups to be familiar with source code and functionality of the application. One group was a control group which was assigned to non-refactoring code and the other group was an experimental group which was assigned to a refactored code. After that a question paper was distributed to participants and 30 minutes were provided to answer the questions. At the end of the experiment, question papers were evaluated and marks were recorded for the analysis.

In order to analyze the impact of refactoring on changeability next step of the experiment was carried out. Source codes with randomly inserted bugs were provided to both experimental and controlled groups. Error descriptions were provided for semantic errors. Participants were worked on fixing bugs and 90 minutes of time frame was provided to fix the bugs. Time used to fix bugs was recorded as data for analysis.

- Step 2:

In order to measure resource utilization; memory consumption of software application to execute one selected task was measured. As stated in [23] memory utilization is a one attribute for predicting the utilization of hardware. To measure time behavior task execution time was measured [23]. When selecting tasks, a piece of code which is mostly affected by applied refactoring techniques was selected as task. Programs were simulated to execute automatically 1000 times in order to collect accurate figures related to execution time and memory consumption during the selected task execution.

IV. Analysis of Data

This section provides a summary of the data collection and an analysis of the impact of refactoring using external measures. The statistical analysis of experiment results and research findings are discussed within this section.

As the research is quantitative and involves ratio data, parametric statistical test was used for hypothesis testing. When the sample size was less than 30, t-distribution was used for hypothesis testing. And when the sample size is greater than 30, z-tests was employed to test difference between two means.

A. Analysis of the individual impact of Refactoring Techniques separately

- Data analysis for Analyzability

Analyzability was measured by using marks obtained by each group member for the given question paper as explained in previous section. The time duration for question paper was 30 minutes and final mark was given out of 10. Table 1 summarized the mean values for each refactoring technique.

Table 1 Mean Values for Analysability (Marks obtained) for each Refactoring Technique

Refactoring Technique	Control Group	Experimental Group
Introduce Local Extension	9.33	8.67
Duplicate Observed Data	8.67	8.67
Replace Type Code with Subclasses	9.33	8.33
Replace Type Code with State/Strategy	8	8.67
Replace Conditional with Polymorphism	6.67	9.67
Introduce Null Object	5.67	8.33
Extract Subclass	6	6
Extract Interface	7	7
Form Template Method	8.33	8
Push Down Method	9	8.67

A common hypothesis which is being tested under Analyzability for each refactoring technique is that "analyzability of refactored code is higher than non-refactored code". Table 2 summarized the results of hypothesis testing for each refactoring technique.

Table 2 Summary of Hypotheses Testing Results for Analysability for each Refactoring Techniques

Refactoring Technique	H_0 Reject	H_0 Accept
Introduce Local Extension		*
Duplicate Observed Data		*
Replace Type Code with Subclasses		*
Replace Type Code with State/Strategy		*
Replace Conditional with Polymorphism	*	
Introduce Null Object		*
Extract Subclass		*
Extract Interface		*
Form Template Method		*
Push Down Method		*

Except one refactoring technique which is "Replace Conditional with Polymorphism", for other refactoring techniques the assumption of better analyzability thus cannot be answered according to hypothesis testing for the mini size code.

- Data analysis for Changeability

The changeability of individual refactoring technique, time needed to fix bugs in minutes was used. Table 3 summarized the experimental results.

Table 3 Summarized Results for Changeability (in Minutes) for each Refactoring Technique

Refactoring Technique	Control Group	Experimental Group
Introduce Local Extension	9	14
Duplicate Observed Data	6.33	12.3
Replace Type Code with Subclasses	12.6	6.67
Replace Type Code with State/Strategy	5	7.67
Replace Conditional with Polymorphism	8.67	13.3
Introduce Null Object	24.6	29
Extract Subclass	22	31
Extract Interface	13.6	10
Form Template Method	9.67	26.3
Push Down Method	4.33	10

Hypothesis which is tested under Changeability for each refactoring technique is that the "changeability of refactored code is easier than non-refactored code". Table 4 summarized results of hypothesis testing for each refactoring technique.

Table 4 Summary of Hypotheses Testing Results for Changeability for each Refactoring Techniques

Refactoring Technique	H_0 Reject	H_0 Accept
Introduce Local Extension		*
Duplicate Observed Data		*
Replace Type Code with Subclasses		*
Replace Type Code with State/Strategy		*
Replace Conditional with Polymorphism		*
Introduce Null Object		*
Extract Subclass		*
Extract Interface		*
Form Template Method		*
Push Down Method		*

The assumption of better changeability for all the refactoring techniques thus cannot be answered according to hypothesis tests; because, there is an insufficient statistical evidence to claim that time spent by experimental group is less than control group. Therefore, the conclusion of better changeability is not facilitated with the mini size source code.

- Data analysis for Time Behavior

The measurement of time behavior related for each refactoring technique was measured by recording task execution time as explained earlier. Results were recorded in milliseconds.

Table 5 Summarized Results for Time Behaviour (in Milliseconds) for each Refactoring Technique

Refactoring Technique	Control Group	Experimental Group
Introduce Local Extension	1.63	1.51
Duplicate Observed Data	138.46	141.39
Replace Type Code with Subclasses	0.04	0.06
Replace Type Code with State/Strategy	0.02	0.03
Replace Conditional with Polymorphism	0.23	0.21

Introduce Null Object	0.0004	0.0009
Extract Subclass	269.29	304.98
Extract Interface	17.27	36.11
Form Template Method	0.23	0.27
Push Down Method	10.36	10.17

A hypothesis which was tested for time behavior is that the "response time of refactored code which is less than non-refactored code". Table 6 summarized the results of hypothesis testing.

Table 6 Summary of Hypotheses Testing Results for Time behaviour for each Refactoring Techniques

Refactoring Technique	H_0 Reject	H_0 Accept
Introduce Local Extension	*	
Duplicate Observed Data		*
Replace Type Code with Subclasses		*
Replace Type Code with State/Strategy		*
Replace Conditional with Polymorphism	*	
Introduce Null Object		*
Extract Subclass		*
Extract Interface		*
Form Template Method		*
Push Down Method	*	

.

Among the evaluated ten refactoring techniques, only three refactoring techniques; "Introduce Local Extension", "Replace Conditional with Polymorphism" and "Push down Method" indicated that there is better time behavior after in refactored code. However, the assumption of better time behavior for the refactored code cannot be answered for the majority of refactoring techniques according to hypothesis testing; because according to the hypothesis test results, there is insufficient statistical evidence to claim a time spent by refactoring code to respond for particular task is less than non-refactored code.

- Data analysis for Resource Utilization

Resource utilization was measured for each selected refactoring techniques by using memory consumption of program while it was executing as explained earlier. Results were recorded in bytes.

Table 7 Summarized Results for Resource Utilization (in bytes) for each Refactoring Technique

Refactoring Technique	Control Group	Experimental Group
Introduce Local Extension	8192.00	8192.00
Duplicate Observed Data	170062.85	165414.53
Replace Type Code with Subclasses	8192.00	8192.00
Replace Type Code with State/Strategy	8192.00	8192.00
Replace Conditional with Polymorphism	8192.00	8192.00
Introduce Null Object	0.00	8192.00
Extract Subclass	7246943.48	7246391.17
Extract Interface	519120.00	519120.00
Form Template Method	8192.00	8192.00
Push Down Method	25742.81	25834.20

A hypothesis which was tested for Resource Utilization is that "efficient utilization of computer Resources which is higher for the refactored code than the non-refactored code". Table 8 summarized the results of hypothesis testing.

Table 8 Summary of Hypotheses Testing Results for Resource Utilization for each Refactoring Techniques

Refactoring Technique	H_0 Reject	H_0 Accept
Introduce Local Extension	-	-
Duplicate Observed Data	*	
Replace Type Code with Subclasses	-	-
Replace Type Code with State/Strategy	-	-
Replace Conditional with Polymorphism	-	-
Introduce Null Object	-	-
Extract Subclass	*	
Extract Interface	-	-
Form Template Method	-	-
Push Down Method		*

Hypothesis testing for resource utilization for both "Duplicate Observed Data" and "Extract Subclass" refactoring techniques indicates better resource utilization. However, hypothesis testing could not be able to carry out for some experimental results due to zero deviation within experimental results. Other experiments are ended up with the result as there is insufficient statistical evidence to claim that better resource utilization in term of memory consumption.

- Summary of Results

Table 9 presents summary of hypothesis testing results. The following symbols are used to indicate the results.

- Null Hypothesis Rejected: '+'
- Null Hypothesis Accepted: '-'
- Hypothesis testing is not applicable: '0'

Table 9 Summary of hypothesis testing results for the effect of each refactoring on code quality using external measures

Refactoring Techniques	Analyzability	Changeability	Time Behavior	Resource Utilization	**H_0 Accepted**	**H_0 Rejected**
Introduce Local Extension	-	-	+	0	**2**	**1**
Duplicate Observed Data	-	-	-	+	**3**	**1**

Replace Type Code with Subclasses	-	-	-	0	**3**	**0**
Replace Type Code with State/Strategy	-	-	-	0	**3**	**0**
Replace Conditional with Polymorphism	+	-	+	0	**1**	**2**
Introduce Null Object	-	-	-	0	**3**	**0**
Extract Subclass	-	-	-	+	**3**	**1**
Extract Interface	-	-	-	0	**3**	**0**
Form Template Method	-	-	-	0	**3**	**0**
Push Down Method	-	-	+	-	**3**	**1**

Through the hypothesis testing results, it can be noticed that except refactoring technique "Replace Conditional with Polymorphism", all the other refactoring techniques show higher number of quality deteriorates than quality improvements.

B. Analysis of the overall impact of selected Refactoring Techniques

- Data Analysis for Analyzability

Analyzability was measured by using marks obtained by each group member for the given question paper. Same question paper which contained 15 multiple choice and short answer questions was distributed to both controlled and experimental groups. The time duration for question paper was 30 minutes and final mark was given out of 15. Hypothesis which was tested for Analyzability is that "analyzability of refactored code is higher than non-refactored code". Table 10 summarized results of hypothesis testing.

Table 10 Hypothesis test results for Analysability

Level of Significance	0.05
Controlled Group	
Sample Size	10
Sample Mean	7.1
Sample Standard Deviation	3.6
Experimental Group	
Sample Size	9
Sample Mean	6.63
Sample Standard Deviation	2.13
t Test Statistic	0.344524
p-Value	0.466775
Do not reject the null hypothesis	

The assumption of better analyzability cannot be answered according to hypothesis test results; because there is insufficient statistical evidence to claim marks obtained by experimental group is higher than control group. In fact it is lesser in experimental group. Therefore, it can be stated that refactoring does not significantly affect analyzability of small scale code.

- Data Analysis for Changeability

The measurement of changeability, which consisted of a random insertion of two non-syntactical errors and one new requirement, was measured by using time needed to fix bugs in minutes. Hypothesis which was tested under Changeability is that "changeability of refactored code is easier than non-refactored code". Table 11 summarized results of hypothesis testing.

Table 11 Hypothesis Test Results for Changeability

Level of Significance	0.05
Controlled Group	
Sample Size	10
Sample Mean	59
Sample Standard Deviation	26.27
Experimental Group	
Sample Size	10
Sample Mean	77
Sample Standard Deviation	27.72
t Test Statistic	-1.57325
p-Value	0.933464
Do not reject the null hypothesis	

The assumption of better changeability thus cannot be answered according to hypothesis testing; because, there is insufficient statistical evidence to claim that time spent by experimental group is less than control group. Therefore, it can be stated that refactoring does not significantly affect changeability of small scale code.

- Data analysis of Time Behavior

The measurement of time behavior was measured by recording task execution time. Piece of code which is highly affected by refactoring treatment was selected and the task which is related to that code segment was selected for testing. Both pre and post refactored programs were modified to execute 1000 times automatically. Results were recorded in milliseconds. Outliers were detected from 1000 sample size from both samples. A hypothesis which was tested for Time Behavior is that "response time of refactored code is less than non-refactored code". Table 12 summarized results of hypothesis testing.

Table 12 Hypothesis Test Results for Time Behaviour

Level of Significance	0.05
Original Code	
Sample Size	994
Sample Mean	61.18
Population Standard Deviation	21.22
Refactored Code	
Sample Size	985
Sample Mean	75.71
Population Standard Deviation	20
Z-Test Statistic	-15.7109
p-Value	1
Do not reject the null hypothesis	

The assumption of better time behavior of refactored code thus cannot be answered according to hypothesis testing; because, there is insufficient statistical evidence to claim that

task execution time for refactored code is less than code without refactoring. Therefore, the conclusion of better time behavior is not facilitated by refactoring.

- Data analysis for Resource Utilization

Resource utilization was measured by using memory consumption of program while it is executing. Piece of code which is highly affected by refactoring treatment was selected and the task which is related to that code segment was selected for testing. Both pre and post refactored programs were changed to execute 1000 time automatically. Results were recorded in bytes. Outliers were detected from 1000 sample size from both samples. A hypothesis which was tested for Resource Utilization is "efficient utilization of computer Resources is higher for refactored code than non-refactored code". Table 13 summarized results of hypothesis testing.

Table 13 Hypotheses Testing results for Resource Utilization

Level of Significance	0.05
Original Code	
Sample Size	1000
Sample Mean	370970.4
Population Standard Deviation	159046.9
Refactored Code	
Sample Size	1000
Sample Mean	377310.3
Population Standard Deviation	162510.2
Z-Test Statistic	-0.88169
p-Value	0.811027
Do not reject the null hypothesis	

The assumption of better resource utilization of refactored code thus cannot be answered according to hypothesis testing; because according to the hypothesis test results, there is insufficient statistical evidence to claim a minimum memory allocation for refactored code than non-refactored code. Therefore, the conclusion of better resource utilization is not facilitated by refactoring.

- Summary of Results

Table 14 shows the summary of hypothesis testing results of the impact of refactoring on code quality measured by using external measures. In the table symbols are represented as follows.

- Improvement: '+'
- Deteriorate: '-'
- No impact: '0'

Table 14 Summary of the effect of refactoring on code quality using external measures

Analyzability	Changeability	Time Behavior	Resource Utilization	**No. Deteriorates**	**No. Improvements**
-	-	-	-	**4**	**0**

Here it can be noticed that none of the external measures show improvements in code quality when all the selected refactoring techniques are applied together.

V. Research Findings and Discussion

Impact of refactoring on code quality improvement using external measures were measured using four sub quality factors defined in ISO 9126 quality model. Firstly, the individual impact of selected refactoring techniques on code quality was measured. Summarized results were presented in Table 15 and for each refactoring technique the percentage of quality improvements, unchanged and deteriorates were presented.

Table 15 Summary of analysis of refactoring techniques using external measures

Refactoring Techniques	Deteriorates	Unchanged	Improvements
Introduce Local Extension	50%	25%	25%
Duplicate Observed Data	75%	0%	25%
Replace Type Code with Subclasses	75%	25%	0%
Replace Type Code with State/Strategy	75%	25%	0%
Replace Conditional with Polymorphism	25%	25%	50%
Introduce Null Object	100%	0%	0%
Extract Subclass	75%	0%	25%
Extract Interface	75%	25%	0%
Form Template Method	75%	25%	0%
Push Down Method	75%	0%	25%

Except "Replace conditional with polymorphism" which is having the highest percentage of quality improvement, all the other refactoring techniques have a high percentage of deteriorate of quality according to the results of analysis. Among them "Introduce null object" have the highest percentage of deteriorate of quality according to Table 15.

For each external measure, the percentage of improvements, unchanged and deteriorates were calculated from tested ten refactoring techniques.

Table 16 Summary of effect of refactoring on external measures – Analysis of each refactoring techniques

External Measure	Deteriorate	Unchanged	Improvement
Analyzability	90%	0%	10%
Changeability	100%	0%	0%
Time Behavior	70%	0%	30%
Resource Utilization	10%	60%	30%

From the results summarized in Table 16, it can be concluded that there is a significant negative effect on code analyzability, changeability and time behavior. However, resource utilization of refactored code is unchanged when it compare with same non-refactored code.

In order to further analyze the results of first experiment, second experiment was executed to identify the overall impact of selected refactoring techniques on code quality. Hypothesis test results indicate that there is deteriorate of code quality in refactored code than non-refactored code. Table 17 summarized the findings of analysis of the overall impact of refactoring on code quality.

Table 17 Summary of effect of refactoring on external measures – Overall Analysis of refactoring techniques

External Measure	Deteriorate	Unchanged	Improvement
Analyzability	*		
Changeability	*		
Time Behavior	*		
Resource Utilization	*		

When results of overall analysis from Table 17 and aggregated results of analysis of each refactoring technique from Table 16 are compared, the results for analyzability, changeability and time behavior are similar to each other. Therefore, by using overall analysis and analysis of each refactoring technique, it can be concluded that code analyzability, changeability and time behavior deteriorate after applying ten refactoring techniques which was used for this study.

According to the analysis of individual refactoring techniques, the new ranking for selected 10 refactoring techniques can be presented. Here in Table 18 it presents comparison between Shatnawi and Li's [20] ranking and new ranking proposed with this study.

Table 18 Proposed Ranking for Refactoring Techniques According to the impact on external measures

Proposed Ranking	Refactoring Technique	Shatnawi and Li's [20] Ranking
1	Replace Conditional with Polymorphism	5
2	Introduce Local Extension	1
3	Duplicate Observed Data	2
4	Extract Subclass	7
5	Push Down Method	10
6	Replace Type Code with Subclasses	3
7	Replace Type Code with State/Strategy	4
8	Extract Interface	8
9	Form Template Method	9
10	Introduce Null Object	6

From the analysis of four external measures "Replace Conditional with Polymorphism" ranked in the highest as having a high percentage of improvement in code quality. "Introduce Null Object" was ranked as worst which is having the highest percentage of deteriorate of code quality.

VI. Conclusion And Future Works

The main objective of this study was to assess the impact of refactoring on code quality improvement in software maintenance. In order to achieve that, the impact of refactoring was assessed using external measures namely; analyzability, changeability, time behavior and resource utilization. Experimental research approach was used to assess the ten selected refactoring techniques.

When analyzing all the refactoring techniques together and separately, analyzability resulted as deteriorate the code analyzability after refactoring the source code. Further, the analysis of refactoring techniques together and analysis of refactoring techniques separately, for changeability it indicates a negative impact on code changeability by refactoring. Results of time behavior indicate negative impact of refactoring. When analyzing all the refactoring techniques together, resource utilization indicates that the efficient utilization of computer resources is low for refactored code than non-refactored code. However, when analyzing each refactoring techniques separately the summarized result indicates that the efficient utilization of computer resources is kept unchanged for both refactored code and non-refactored code.

According to the results of individual analysis of refactoring techniques, the most beneficial refactoring technique among evaluated 10 refactoring techniques is reported as "Replace Conditional with Polymorphism".

Finally, according to the results of both overall analysis and individual analysis of refactoring it can be stated that refactoring does not improve the code analyzability and code changeability in small size applications. Further refactoring does not support better resource utilization and refactoring does not have better time behavior while executing small scale source code.

The results of this study indicate that there is further need of addressing the impact of refactoring. Refactoring techniques used in this study were selected from the ranking done by previous study [20]. Therefore, in the future it is better to conduct a study to find refactoring techniques which are commonly used in industry by a survey. Then do the analysis of the impact of those commonly used refactoring techniques. That will be more advantageous to the software development industry rather than selecting refactoring techniques subjectively. Further, it will be better that if the same experimental setup can be execute in industry environment with the industry experts and with the industry level matured source code. Then the outcome of this study can be able to validate against the outcome of that study.

VII. ACKNOWLEDGEMENTS

Special thanks go to all the participants of the experiment for their contribution of valuable time and effort.

REFERENCES

[1] International Standards. (2001). ISO/*IEC 9126-1 Standard*. [Online]. Available: http://webstore.iec.ch/preview/infoisoiec91261%7Bed1.0%7Den.pdf.

[2] R Pressman, *Software Engineering: A Practitioner's Approach,* 6th edn, McGraw-Hill, 2005

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

[4] T. Mens and T.A. Tourwé, "Survey of Software Refactoring", *IEEE Trans. on Software Engineering*, vol. 30, no. 2, pp. 126-139, 2004.

[5] K. Jussi. (2010). Software Maintenance Costs. [Online]. Available: http://users.jyu.fi/~koskinen/smcosts.htm.

[6] M. Fowler, *Refactoring Improving the Design of Existing Code*, Addison-Wesley, 1999.

[7] B.D. Bois and T. Mens, "Describing the impact of refactoring on internal program quality", in *Proc. of the International Workshop on Evolution of Large-scale Industrial Software Applications, Amsterdam*, The Netherlands, 2003, pp. 37-48.

[8] Y. Kataoka et al., "A quantitative evaluation of maintainability enhancement by refactoring", in *Proc. of the IEEE International Conference on Software Maintenance*, Montreal, Quebec, Canada, 2002.

[9] D. Wilking, U. Khan and S. Kowalewski, "An empirical evaluation of refactoring", *e- Informatica Software Engineering Journal*, vol. 1, pp. 27-42, 2007.

[10] T. Mens et al.," Refactoring: Current Research and Future Trends", *Electronic Notes in Theoretical Computer Science*, vol.80, no.3, 2003.

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

[12] B.D. Bois et al., "Refactoring – improving coupling and cohesion of existing code", in *Proc. of 11th Working Conference on Reverse Engineering (WCRE'04)*, 2004, pp. 144–151.

[13] S. H. Kannangara and W.M.J.I. Wijayanayake, "Measuring the Impact of Refactoring on Code Quality Improvement Using Internal Measures", *In Proc. of the International Conference on Business & Information*, Sri Lanka, December 2013.

[14] B. Geppert et al., "Refactoring for changeability: a way to go", in *Proc. of 11th IEEE International Software Metrics Symposium (METRICS'05),* Como, Italy, 2005.

[15] S. H. Kannangara and W.M.J.I. Wijayanayake, "Impact of Refactoring on External Code Quality Improvement: An Empirical Evaluation*", In Proc. of International Conference on Advances in ICT for Emerging Regions*, Sri Lanka, December 2013.

[16] S. H. Kannangara and W.M.J.I. Wijayanayake, "Impact of Refactoring on Code Quality Improvement in Software Engineering*", In Proc. of 2nd National Conference on Technology and Management*, Sri Lanka, January 2013.

[17] F. Dandashi and D.C. Rine, "A Method for Reusability of Object-Oriented Code Using a Validated Set of Automated Measurements", in *Proc. of 17th ACM Symposium on Applied Computing (SAC 2002)*, Madrid, 2002.

[18] R. Moser et al., "Does Refactoring Improve Reusability?", in *Proc. of 9th International Conference on Software Reuse (ICSR'06)* , 2006, pp.287–297.

[19] R. Moser et al., "A case study on the impact of refactoring on quality and productivity in an agile team", *in Proc. of the Central and East-European Conference on Software Engineering Techniques*, Poznan, Poland, 2007.

[20] R. Shatnawi and W. Li., "An Empirical Assessment of Refactoring Impact on Software Quality Using a Hierarchical Quality Model", *International Journal of Software Engineering and Its Applications*, vol. 5, no. 4, 2011.

[21] R. E. Al-Qutaish, "Quality Models in Software Engineering Literature: An Analytical and Comparative Study", *Journal of American Science*, vol. 6, no. 3, pp. 166-175, 2010.

[22] Hani. (2009) Placebo Effect [Online]. Available: http://www.experiment - resources.com/placebo-effect.html.

[23] (2012) [Online]. ISO 9126 Metrics. Available: http://www.rockynook.com/samples /97/ISO_9126_Metrics.pdf.