2013 International Conference on Advances in ICT for Emerging Regions (ICTer): 060 - 067

Impact of Refactoring on External Code Quality Improvement: An Empirical Evaluation

S.H. Kannangara [#1], W.M.J.I. Wijayanayake [#2]

[#]*Department of Industrial Management, Faculty of Science, University of Kelaniya, Sri Lanka*

[1]sandeepa.kannangara@gmail.com, [2]janaka@kln.ac.lk

***Abstract*— Refactoring is the process of improving the design of the existing code by changing its internal structure without affecting its external behaviour, with the main aims of improving the quality of software product. Therefore, there is belief that refactoring improves quality factors such as understandability, flexibility, and reusability. Moreover, there are also claims that refactoring yields higher development productivity. 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 impact of each refactoring technique was assessed based on external measures namely; analysability, changeability, time behaviour and resource utilization.

After analysing the experimental results, among the tested ten refactoring techniques, "Replace Conditional with Polymorphism" ranked in the highest as having high percentage of improvement in code quality. "Introduce Null Object" was ranked as worst which is having highest percentage of deteriorate of code quality.

Keywords— **Refactoring, Software maintenance, Analysability, Changeability, Time behaviour, Resource Utilization, ISO 9126**

I. INTRODUCTION

Any useful software system requires constant evolution and change. As the software system is enhanced, modified and adapted to new requirements, the code become more complex and drifts away from its original design. Because of this, the major part of the total software development cost is devoted to software maintenance. Maintenance of software is reported as a serious cost factor [1] and as stated in [2], over 90% of the software development cost is for software maintenance.

While software system is evolving, maintaining the software quality is one of the vital factors in software maintenance process. Software maintenance best practices are arising in this case with the purpose of a better evolution of software while preserving the quality of software systems. Quality software are robust, reliable and easy to maintain, and therefore reduces the cost of software maintenance [3]. Developers and designers always strive for quality software. Software quality can be described as the conformance to functional requirements and non-functional requirements, which are related to characteristics described in the ISO-9126 standard namely reliability, usability, efficiency, maintainability and portability [4].

One solution proposed to reduce the software maintenance effort is software code refactoring [5] which is a method to continuous restructure code according to implicit micro design rules. According to the Fowler's definition in [5], 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 behaviour. Refactoring is by definition supposed to improve the maintainability of a software product, but its effect on other quality aspects is unclear. Therefore, there are hot and controversial issues in refactoring.

As stated in [1] refactoring is assumed to positively affect non-functional aspects, likely extensibility, modularity, reusability, complexity, maintainability, and efficiency. Recently [6] 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 [1]. They consist of additional 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 ([7], [8]). Even though 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 [9]. And also there is lack of studies on identification of most beneficial refactoring techniques among available large number of refactoring techniques, in terms of improvement in software quality. As mentioned in [10] 'effect of a refactoring on the software quality' is a one of the open issues that remain to be solved.

Altogether, the real advantages of refactoring are still to be fully assessed. As regards quality, it appears to be a convergence of positive remarks, still, without solid quantification. In addition 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 program refactoring before applying it. Without knowing which refactoring technique will be more beneficial in terms

978-1-4799-1276-6/13/$31.00 ©2013 IEEE

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 purpose of this study was to conduct a set of experiments on the field of 'code refactoring' in order to identify the refactoring techniques which have the highest impact on code quality improvement that can help software developers in order to select the most effective refactoring techniques.

The reminder of this paper structured as follows: Section 2 provides a summary of relevant literature which are addressed the relationship between refactoring and software quality and maintenance. Experimental design 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. Few researches quantitatively evaluated whether refactoring indeed improves quality (e.g. [7], [8]).

Among them, significant number of researchers quantitatively evaluated the impact of refactoring using internal quality software attributes. Bois and Mens [6] proposed a technique using metrics to analyse 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 only on three refactoring methods. But they did not provide any experimental validation in an industrial environment. The results in [6]'s work showed both positive and negative impacts on the studied measures. Stroggylos and Spinellis [10] analysed 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. [11] 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 and came up with results that the effect of refactoring on coupling and cohesion measures ranged from negative to positive.

Some of the other studies took the approach of assessing the refactoring effects on external software quality attributes. Geppert et al. [12] 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 as investigated by [8] through an empirical evaluation. Maintainability was tested by randomly inserting 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.

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. Kataoka et al. [7] proposed coupling metrics as a quantitative evaluation method to measure the effect of refactoring on program maintainability. For the purpose of validation they analysed a C++ program for two refactoring techniques: Extract Method and Extract Class which developed by a single developer, but 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. [13] proposed a methodology to assess whether or not 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 [14]. 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 analysed by [15]. 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, using software matrices based on metric interpretation of [14], the effect of refactoring on different external quality attributes (Adaptability, Maintainability, Understandability, Reusability, Testability). But this study didn't prove that refactoring improves external quality of the software. Shatnawi and Li [16] studied the effect of software refactoring on software quality. They have conducted the study on a larger number of refactoring techniques (43 refactoring) using a Quality Model for OO Design (QMOOD) on four quality factors measured indirectly using nine different software measures. 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 analysing 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 analysing 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 [16] all the other studies used only less than ten refactoring techniques for their evaluation. Most of them did not consider any valid fact when selecting refactoring techniques for their study.
- Finally, as most of the studies did not evaluate large number of refactoring techniques, they couldn't be able to identify the most beneficial refactoring techniques among catalogue of large number of refactoring techniques.

To overcome the above issues, this study was conducted on a considerable number of refactoring techniques and only focused on external quality factors selected from ISO quality model. And also the identification of most beneficial refactoring techniques among selected refactoring was another goal of this research.

III. Experimental Design

As the objective of this study was to quantitatively assess the impact of each selected refactoring techniques separately using external measures in order to identify refactoring techniques which have the highest impact on external code quality improvement, quantitative research approach is more preferable.

Experiential evidence of 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 impact of refactoring on code quality.

The general approach followed by experiment was consisting two groups per each refactoring techniques using the same application developed by using C#.net. For each refactoring technique one group was assigned refactored code using selected refactoring technique while the rest will be assigned source code without refactoring. The assignment to a treatment and control groups were done randomly.

A. Selected Refactoring Techniques

Fowler [4] proposed 72 refactoring techniques in his catalogue of refactoring. Because of time limitations and size of source code, it is not possible to evaluate all the refactoring techniques.

Among the studies which have evaluated the impact of refactoring, the most recent study [16] present large evaluation of 43 refactoring techniques among 72 refactoring techniques in Fowler's [4] catalogue. Evaluated refactoring techniques were ranked according to the impact of code quality. Therefore, for this study, ten refactoring techniques were selected from [16]'s study which were ranked as having a high impact.

Selected Refactoring Techniques are follows:

- 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 Development Environment

Refactoring is a technique which is mainly related to 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. [7], [16]).

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

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.

It can be noticed that most of the previous studies were limited only to a few external quality attributes as described in section II. In this research, the main consideration was use of external quality attributes to get a precise indication of whether or not software quality can be improved by refactoring. As stated in [20], 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 [4] 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: A set of attributes that bears the effort needed to make specified modifications. Following sub characteristics were tested in this study [4].
   i. Analysability
   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 [4].
   iii. Resource Utilization
   iv. Time behaviour

Other quality factors in ISO quality model have to be excluded from this study. Excluded the functionality factor, because refactoring does not change the behaviour of systems, rather it changes the internal characteristics of systems without changing functionality. Usability factor was excluded, because it is more implementation oriented. Usability indicates how easy it is to learn and use the software. Reliability is also implementation oriented quality factor. Reliability is an attribute that can only be estimated by actually running the software several times 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 how easy it is to migrate the software to a different hardware or an Operating system. But 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 in subjects given control [19].

2. Dependent Variables

Dependent variables for this experiment are,

- Marks obtained for question paper
- Time need to fix bugs
- Execution Time
- Memory Consumption

E. Research Hypothesis

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

- Analysability
  H_0A: Analysability of refactored code is lower than non-refactored code.
  H_1A: Analysability 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 Behaviour
  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 sixty students. 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 Department of Industrial Management, Faculty of Science, 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 done in order to identify student's familiarity of C#.Net and Object Oriented Concepts: Online questionnaire was designed to gather responses.

After collecting both data, students' results and responses were 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.

Due to the availability of limited resources at Undergraduate laboratories and controlling of large groups was not possible with available human resources, group size was decided as 3 members per one group. Therefore, the required number of participants for the experiment was 60.

G. General Procedure

The general procedure for experiment was mainly carried out in two steps for each refactoring technique. Therefore, ten experiments have to be carried out by using the same procedure.

The first step of each experiment was done with controlled and experimental groups which consist of 60 students. The second step for each experiment was carried out in a software testing environment, in order to collect resource utilization and time behaviour measures.

In order to apply each refactoring technique separately, mini size projects were selected as the source code. As stated in sub section B, the code developed by using C#.net was used for the experiment. Bad smells were identified and suitable refactoring technique was applied to each source code. Finally the ten different refactored source codes and the original source codes of those were available for the experiment.

- 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 analysability. Initially several minutes were provided to both groups to be familiar with source code and program. One group was a control group which was assigned to code without refactoring 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 questions. At the end of the experiment, question papers were evaluated and marks were recorded for the analysis.

In order to analyse the impact of refactoring on changeability next 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 for fixing 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 and to measure time behaviour task execution time was measured [19]. 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 each refactoring technique using external measures. The statistical analysis of experiment results and research findings are discussed within this section.

A. Data analysis for Analysability

Analysability was measured by using marks obtained by each group member for the given question paper. The same question paper which contained 10 multiple choice and short answer questions were distributed to both controlled and experimental groups.

The time duration for question paper was 30 minutes and final mark was given by out of 10. The results were gathered from both groups which contains three members per group and the corresponding summarized results for each refactoring technique is aggregated into the Table 1.

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

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

A common hypothesis which is being tested under Analysability for each refactoring technique was the analysability of refactored code is higher than non-refactored code. As the size of one group is 3, which is less than 30, t-distribution was used. Therefore, as the statistical test, pooled-Variance t-test for the difference between two means was employed.

Table 2 summarized the results of hypothesis testing for each refactoring technique.

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		*

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

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

B. Data analysis for Changeability

To measure the changeability of each refactoring technique, which consisted of a random insertion of one non-syntactical errors and one new requirement, time needed to fix bugs in minutes was used. The errors were created by interchanging code pieces and assigning some invalid values for variables.

The results were gathered from both groups which contains 3 members per group and the corresponding results were aggregated into the Table 3.

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

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

Hypothesis which is tested under Changeability for each refactoring technique was the changeability of refactored code is easier than non-refactored code. As the size of one group is 3, which is less than 30, t-distribution was used. So as a statistical test z-test for the difference between two means was employed.

Table 4 summarized results of hypothesis testing for each refactoring technique.

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		*

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

The assumption of better changeability for all the refactoring techniques thus cannot be answered according to hypothesis tests; because according to the hypothesis test results, there is an insufficient statistical evidence to claim a time spent by experimental group is less than control group. Therefore, the conclusion was better changeability is not facilitated by each refactoring technique.

C. Data analysis for Time Behaviour

The measurement of time behaviour related for each refactoring technique was measured by recording task execution time. Piece of code which is highly affected by refactoring treatment was selected. Both refactored and non-refactored programs were simulated to execute 1000 time automatically. Results were recorded in milliseconds.

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

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

A hypothesis which was tested for time behaviour is the response time of refactored code which is less than non-refactored code. As the size of one sample is nearly 1000, which is greater than 30, and by assuming population is normally distributed, z-distribution was used. . Therefore, as a statistical test, z-test for the difference between two means was employed.

Table 6 summarized the results of hypothesis testing which are done to evaluate each refactoring technique.

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	*	

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

The assumption of better time behaviour for the refactored code thus 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.

D. Data analysis for Resource Utilization

Resource utilization was measured for all selected refactoring techniques by using memory consumption of program while it was executing. Piece of code which is highly affected by a refactoring treatment was selected and the task which is related to that code segment is selected for testing. Both refactored and non-refactored programs were simulated

to execute 1000 times automatically. Results were recorded in bytes.

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

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

A hypothesis which was tested for Resource Utilization was the efficient utilization of computer Resources which is higher for the refactored code than the non-refactored code. As the size of one sample is nearly 1000, which is greater than 30, and by assuming population is normally distributed, z-distribution was used. Therefore as a statistical test, z-test for the difference between two means was employed.

Table 8 summarized the results of hypothesis testing.

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		*

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

Hypothesis testing for resource utilization for both "Duplicate Observed Data" and "Extract Subclass" refactoring techniques indicates better resource utilization.

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.

V. Research Findings and Discussion

Using external measures selected refactoring techniques were analysed for the impact on each external measure. Summarized results were presented in Table IX and for each refactoring technique the percentage of quality improvements, unchanged and deteriorates were presented.

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%

Table 9: Summary of analysis of refactoring techniques using external measures

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 the Table 9.

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

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

Table 10: Summary of effect of refactoring on external measures – Analysis of each refactoring techniques

From the results summarized in Table 10, it can be concluded that there is a significant negative effect on code analyzability, changeability and time behavior.

Wilking et al. [8] did analysis of the impact of refactoring on code maintainability, modifiability and memory consumption. Maintainability and modifiability were negatively affected by refactoring treatment according to their findings. Those measures are more similar to external measures: analyzability and changeability used in this study. Thus there is a similarity in the results of this study and their study for those of external measures. But for memory

consumption [8] got a positive result which is different from the results got here. That was the only study which was analyzed external measure similar to this study.

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 and ten selected refactoring techniques were used to analyze the impact of each refactoring technique.

The hypothesis testing results indicates that the analysability of refactored code is lower than non-refactored code for all the tested refactoring techniques except for "Replace conditional with polymorphism". In addition the changeability of refactored code is difficult than non-refactored code for all the tested refactoring techniques. Moreover, response time of refactored code is longer than non-refactored code for majority of tested refactoring techniques except for "Introduce local extension". "Replace conditional with polymorphism" and "Push down method". However, the efficient utilization of computer resources is low for refactored code than non-refactored code only for "Push down method".

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.

The one argument that can come against the experiment is that expert developers would constitute a much better evaluation. Their knowledge on better system design might change the experimental results. Therefore, this can be considered as a limitation of this study.

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 [16]. 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 analyse the impact of those commonly used refactoring techniques will be more advantageous to the software development industry rather than selecting refactoring techniques subjectively. And also it had better that 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 validated 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] T. Mens and T.A. Tourwé, "Survey of Software Refactoring", *IEEE Trans.on Software Engineering*, vol. 30, no. 2, pp. 126-139, 2004.

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

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

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

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

[6] 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.

[7] 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.

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

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

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

[17] F. Simon et al., "Metrics based refactoring", in *Proc. of the Fifth European Conference on Software Maintenance and Reengineering*, Lisbon , Portugal, 2001, pp. 30–39.

[18] (2012) [Online]. ISO 9126 Metrics. Available: http://www.rockynook.com/samples /97/ISO_9126_Metrics.pdf.

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

[20] 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.