EFFECT OF REFACTORING ON SOFTWARE QUALITY

Noble Kumari[1] and Anju Saha[2]

[1]USICT, Dwarka, Delhi, India
noblevashishta_127@yahoo.com
[2]USICT, Dwarka, Delhi, India
anju_kochhar@yahoo.com

ABSTRACT

Software quality is an important issue in the development of successful software application. Many methods have been applied to improve the software quality. Refactoring is one of those methods. But, the effect of refactoring in general on all the software quality attributes is ambiguous.

The goal of this paper is to find out the effect of various refactoring methods on quality attributes and to classify them based on their measurable effect on particular software quality attribute. The paper focuses on studying the Reusability, Complexity, Maintainability, Testability, Adaptability, Understandability, Fault Proneness, Stability and Completeness attribute of a software .This, in turn, will assist the developer in determining that whether to apply a certain refactoring method to improve a desirable quality attribute.

KEYWORDS

Metrics, Refactoring, Attributes & External software quality attributes.

1. INTRODUCTION

Refactoring is defined as the "process of improving the design of existing code by changing its internal structure without affecting its external behavior" [7, 8].The poorly designed code is harder to maintain, test and implement and hence the quality of software degrades. The basic goal of refactoring is the safe transformation of the program to improve the quality. The benefit of undertaking refactoring includes improvement of external software quality attributes.

In software program the word “smell” means potential problem in the code. In the refactoring cycle as the smell is found, refactoring methods are applied and code is improved. The cycle continues till we find the maximum efficient code [8].

The external software quality attributes like reusability, complexity, maintainability, testability and performance are dependent of the software metrics. Software metrics are used to predict the value of the software quality attributes. A large number of software metrics have been proposed which are quantifiable indicators of external quality attributes[22].The value of internal software quality metrics like Coupling Factor (CF), Lack of Cohesion of Method (LCOM), Depth of Inheritance Tree (DIT), Weighted Method per Class (WMC), Lines of Code (LOC) and Cyclomatic Complexity (Vg) are desired to be lower in a system whereas Attribute Hiding Factor (AHF) and Method Hiding Factor (MHF) are desired to be higher [20].

Refactoring changes the value of software metrics and hence the software quality attribute. Not all the refactoring methods improve the software quality, so there is need to find out the refactoring methods which improve the quality attributes [6].

The aim of this paper is to find out the effect of refactoring methods on the software metrics. From the relation between the software metrics and external quality attributes direct relation between refactoring methods and software quality attributes is derived.

The paper analyzes the effect of refactoring on the software quality attributes. The classification of refactoring methods is done for particular desired quality attributes and metrics set.

The study also shows that refactoring does not ensure to improve the software quality always. It has to compensate with some attributes to improve the other.

This paper is organized as follows. Section 2 describes the literature review. Section 3 and 4 explains about the research data and refactoring methods respectively. Section 5 explains about the analysis and result of refactoring methods. Section 6 and 7 explains the threats to validity and Conclusion respectively.

2. RELATED WORK

The goal of this paper is to find the effect of refactoring on the external software quality attributes, using software metrics. In this section we review the study of various researchers on the effect of refactoring on software quality attributes.

Cinneide, Boyle and Moghadam [1] studied the effect of automated refactoring on the testability of the software. The aim is to find the refactoring method which improves the cohesion metric and hence the testability of the software. Code-Imp platform is explored for the refactoring purpose and available metrics in the tool are applied. The survey is done with the volunteers where further testing is required to validate that automated refactoring improves the testability of the software.

Sokal, Aniche and Gerosa [2] took data from Apache software and applied refactoring on it. The authors randomly selected the fifty refactoring methods. They classified them in two groups according to their effect on cyclomatic complexity and analyzed the change in code after refactoring. Their studies show that refactoring does not necessarily decrease the cyclomatic complexity but increases the maintainability and readability of the program.

Alshayeb [6] assess the effect of refactoring on the external software quality attributes. The quality attributes taken were Adaptability, Maintainability, Understandability, Reusability and Testability. Code for refactoring is taken from the open source UMLTool,RabtPad and TerpPaint. The author applied different types of refactoring on the code and studied the effect of refactoring on the software metrics. From the relation between the software metrics and external quality attributes, the effect of refactoring is studied. The author found the inconsistent trend in the relationship of refactoring method and external quality attributes.

Elish and Alshayeb [3] studied effect of refactoring on testability of software. They used five refactoring methods: Extract Method, Extract Class, Consolidated Conditional Expression, Encapsulate Field and Hide Method. Chidamber and Kemerer metrics suite [17] is used to find the software metric values. The authors concluded that all the refactoring methods they used increase the testability except the Extract Class method.

Kataoka [5] used coupling metrics to find the effect of refactoring on the maintainability of the software. He proposed a quantitative evaluation method to measure the maintainability enhancement effect of program refactoring and helped us to choose the appropriate refactoring.

Stroggylos [28] analyzed the source code version control system logs of some of the popular open source software system. They found the effect of refactoring on the software metrics to evaluate the impact of refactoring on quality. The results found the increase in metric valued of LCOM, Ca and RFC which degrades the software quality. They concluded that refactoring does not always improve the software quality.

Shrivastava [29] presented a case study to improve the quality of software by refactoring. They took open source and with the Eclipse refactoring tool produced three version of refactored code. The results found that the size and complexity of a software decreases with refactoring and hence maintainability increases.

The study to find effect of refactoring on the software quality attributes has a wide scope. Fowler [7] has given 70 types of refactoring methods and each refactoring method can be linked to the various software quality attributes. So, our focus is to find the effect of fourteen randomly chosen refactoring methods on the various object oriented metrics and hence on the external software quality attributes.

The following quality attributes will be used in the study:

Maintainability: It is defined as the ease with which modification is made on set of attributes. The modification in the attributes may comprise from requirement to design. It may be about correction, prevention and adaptation [6].
Reusability: It is defined as the reusable feature of the software in the other components or in other software system with little adaptation [6].

Testability: It is defined as the degree to which software supports testing process. High testability requires less effort for testing.

Understandability: It is defined as the ease of understanding the meaning of software components to the user [6].

Fault proneness: Fault Proneness in the programs is more prone to the bugs and malfunctioning of the module.

Completeness: Completeness of the program refers for all the necessary components, resources, programs and all the possible pathways for execution of program [9].
Stability: Stability is defined in terms of ability of the program to bear the risk of all the unexpected modification [23].

Complexity: In an interactive system it is defined as the difficulty of performing various task like coding, debugging, implementing and testing the software.

Adaptability: Adaptability of the software is taken in terms of its ability to tolerate the changes in the system without any intervention from any external resource [26].

3. RESEARCH DATA

The classes used for research data in this paper are from an open source code JHotDraw7.0.6 [10]. Erich Gamma and Thomas Eggenschwiler are the authors of JHotDraw [10]. It has been developed as a quite powerful design exercise whose design is based on some well-known design patterns. We took 120 classes of JHotDraw7.0.6 and applied refactoring methods on it.

The aim of making JHotDraw an open-source project is:

- To refactor and hence enhance the existing code.
- To identify new refactoring and design patterns.
- To set it for an example of a well-designed and flexible framework.

4. REFACTORING METHODS

The refactoring methods applied in this paper are taken from the catalog defined by Fowler [7]. The following refactoring methods are applied [12, 18]:

1. Extract Delegate: This refactoring method allows extracting some of the methods and classes from a given class and added them to newly created class. The refactoring resolves the problem of the class which is big in size and performs much functionality. The name of newly created class is given by the user.

2. Encapsulate field: This refactoring allows modifying the access of data from public to private and generating getter and setter method for that field in the inner class.

3. The Replace Inheritance with Delegation: This refactoring allows removing a class from inheritance hierarchy, while maintaining the functionality of the parent class. In this refactoring a private inner class is made, that inherits the former super class. Selected methods of the parent class are invoked through the new inner class.

4. Replace Constructor with Builder method: The Replace Constructor with Builder refactoring helps hide a constructor, replacing it with the references to a newly generated builder class or to an existing builder class.

5. Extract Interface: Extract Interface is a refactoring operation that allows making a new interface with the members from the existing class, struct and interface.

6. Extract Method**: It** is a refactoring operation that allows creating a new method from the existing members of the class.

7. Push Member Down: The Push Members down refactoring allows in relocating the class members into subclass/sub interface for cleaning the class hierarchy.

8. Move Method: This refactoring allows moving a method from one class to another. The need of moving a method comes when the method is used more in other class than the class in which it is defined.

9. Extract Parameter: The Extract parameter refactoring allows selecting a set of parameters to a method or a wrapper class. The need of the refactoring comes when the number of parameter in a method is too large. The process of refactoring is done by delegate via overloading method also.

10. Safe Delete: The Safe Delete refactoring allows you to safely remove the class, method, field, interface and parameter from the code with making the necessary corrections while deleting.

11. Inline: The Inline Method refactoring allows putting the method's body into the body of its caller method.

12. Static: This refactoring is used to convert a non-static method into a static. This allows the method functionality available to other classes without making the new class instance.

13. Wrap Method Return Value: The Wrap Return Value refactoring allows selecting a method and creating a wrapper class for its return values.

14. Replace Constructor with Factory Method: The Replace Constructor with Factory Method refactoring allows hiding the constructor and replacing it with a static method which returns a new instance of the class.

The tool used for refactoring and studying the values of software metrics is Intellij Idea: IDE for java and a reliable refactoring tool. It knows about code and gives suggestion also as a tip. Refactoring methods referenced from Fowler [7] are available in this tool [12]. All the object oriented metrics can be computed using the tool. The tool gives the module, package, class, project and method level program metrics. It is available and easy to use. Table 1 shows the "wrap method return value" refactoring using the tool.

Table 1. Example of "Wrap Method Return value" Refactoring using the IntelliJ Idea tool.

Before Refactoring	**After Refactoring**
public newadded getScrollPane() { if (desktop.getParent() instanceof JViewport) { JViewport viewPort = (JViewport)desktop.getParent(); if (viewPort.getParent() instanceof JScrollPane) return new newadded((JScrollPane) viewPort.getParent()); } return new newadded(null); }	*public noble getScrollPane() { if (desktop.getParent() instanceof JViewport) {JViewport viewPort = (JViewport)desktop.getParent(); if (viewPort.getParent() instanceof JScrollPane) return new noble(new newadded((JScrollPane) viewPort.getParent())); return new noble(new newadded(null)); }*

In inner class name "noble" is made and then refactoring is performed in the tool.

5. Analysis And Results

The focus of this paper is to find the effect of refactoring methods on the software quality attributes and hence categorized the refactoring methods according to particular quality attributes and software metric domain. The values of object oriented software metrics is found before and after refactoring. The result is analyzed according to the value of the software metrics.

To focus our study on the category of refactoring methods, we set up the following hypothesis. For each hypothesis, H0 represents null hypothesis and H1 represents the alternative hypothesis of H0.

Hypothesis 1
H0: Refactoring does not improve software adaptability.
H1: Refactoring improves the software adaptability.

Hypothesis 2
H0: Refactoring does not improve software maintainability.
H1: Refactoring improves the software maintainability.

Hypothesis 3
H0: Refactoring does not improve software Understandability.
H1: Refactoring improves the software Understandability.

Hypothesis 4
H0: Refactoring does not improve software Reusability.
H1: Refactoring improves the software Reusability.

Hypothesis 5
H0: Refactoring does not improve software Testability.
H1: Refactoring improves the software Testability.

Hypothesis 6
H0: Refactoring does not decrease software Complexity.
H1: Refactoring decreases the software Complexity.

Hypothesis 7
H0: Refactoring does not make software less Fault Proneness.
H1: Refactoring makes the software less Fault Proneness.

Hypothesis 8
H0: Refactoring does not improve software Stability.
H1: Refactoring improves the software Stability.

Hypothesis 9
H0: Refactoring does not improve software Completeness.
H1: Refactoring improves the software Completeness.

For validating all the hypothesis of this paper the relation between the values of software metrics and Refactoring methods is given below in Table 1. Where '↓' shows decrease in the value of metric,'↑' means increase in the value of the metric and '-' shows no change in the value of metric.

Table 2. Relation between Refactoring methods and software quality metrics.

Refactoring Method	**WMC**	**Vg**	**LOC**	**NOM**	**CBO**	**LCOM**	**DIT**	**MPC**	**CCavg**	**AHF**	**AIF**	**CF**	**MHF**	**MIF**	**RFC**
Extract Delegate	↑	↓	↓	↑	↑	↓	↓	↑	↑	↓	↓	↓	↑	↓	↑
Encapsulate Field	↑	↑	↑	↑	↑	↑	↑	↑	↓	↑	↓	–	↓	↑	↑
Inheritance To Delegation	↑	↓	↑	↑	↑	↓	↓	↓	↑	↑	↓	↑	↑	↓	↑
Extract Interface	↓	↓	↑	↑	↑	↑	↑	↑	↑	–	↑	↓	↓	↓	↑
Extract Method	↑	↓	↑	↑	↑	↑	↑	–	↓	–	–	–	↑	↓	↑
Push Method Down	↓	↓	↑	↑	↓	↑	↑	↑	↓	↑	↓	↓	↑	↓	↑
Move Method	↑	↓	↑	–	↑	↑	↑	↑	↑	–	↑	–	↑	–	↑
Extract Parameter	↑	↓	↑	↑	↑	–	↑	–	↓	–	–	–	↑	↓	↑
Safe Delete	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↑	↓	↓	↓
Inline	↑	↑	↑	↑	↑	↑	–	↑	↑	↑	↓	↑	↓	↓	↓
Static	–	↑	↑	↑	–	↑	↑	–	↑	–	↓	–	–	↑	↓
Wrap Method	↓	↓	↑	↑	↓	↓	↓	↓	↓	↑	–	↓	↑	↓	↓

Return Value															
Replace Constructor with factory method	↑	↓	↑	↑	↓	↑	↑	–	↓	–	↓	↓	↑	↑	↓
Replace Constructor with Builder	↓	↓	↑	↑	↓	↓	↓	↓	↓	↑	↓	↓	↑	↓	↓

After Analyzing Table 2, it is concluded that the following methods give desirable result for every metrics [20] and hence improves the quality attribute of software:

1. Wrap Method Return Value
2. Static method

As indicated in hypothesis, we are attempting to find out the refactoring method which improves a particular category of software metrics. The metrics are divided according to the type of impact they make on the software. Table 3 summarizes the relation between metrics and their categories.

Table 3. Relation between metrics and their Categories.

Category	**Attributes**	**Method**	**Coupling /Cohesion**	**Inheritance**
MOOD[27]	AHF, AIF	MHF, MIF ,PF		MIF, AIF
C & K[17]	LCOM	LCOM,WMC, RFC	CBO	DIT
Li and Henry[19]		MPC ,NOM	MPC	

The various refactoring methods show random effect on the metric values. So, we classify the refactoring methods according to the desirable effects they make on the categories of Table 3: attributes, methods, coupling/cohesion and inheritance based metrics. From Table 2 and Table 3 the analysis result is shown in Table 4.

Table 4. Desirable refactoring for the particular category of metrics.

Category	**Refactoring Method**
Attributes	Inheritance to delegation, Wrap return value method and Constructor to Builder
Methods	Wrap Method Return Value
Coupling/Cohesion	Safe Delete ,Replace constructor with builder method , Replace constructor with factory method and Wrap Method Return Value
Inheritance	Extract Delegate, Inline, Safe Delete and Inheritance To Delegation

We used the previously published research work to make the correlation between the software metrics and external quality attributes. We used work of Dandashi [9] to assess the adaptability, maintainability, understandability and reusability quality attributes. The following table summarizes the relationship between software metrics and external quality attributes which can be helpful to find out the direct effect of refactoring on the external software quality attributes.

In this relationship (+) shows positive correlation, the attributes improve as the metric value increases, (-) shows negative correlation, the attributes degrade as the metric value decreases and (0) shows neutral effect.

Table 5. Relation between metrics and external quality attributes.

External Quality	**DIT**	**CBO**	**RFC**	**WMC**	**NOM**	**LOC**	**LCOM**
Adaptability[9,6]	-	-	-	+	0	+	0
Maintainability[22,16,9,25,6]	-	-	-	+	-	+	-
Understandability[9,16,6]	-	-	-	+	0	+	-
Reusability[22,9,16,6]	+	-	-	+	0	+	-
Testability[21,22,6]	-	-	-	-	-	-	-
Complexity[21]	+	+			+	+	+
Fault Proneness[22,24]	+	+	+	+		+	+
Stability[23]	-	-	-	-			-
Completeness[9]	-	-	-	+	-	+	0

To validate the hypothesis, we took the relation between Table 2 and Table 5 and come to the following conclusion:

Table 6. Particular refactoring method for certain quality attributes.

Refactoring Method	**Quality Attribute**
Wrap Return value	Testability
Safe Delete	Adaptability, Understandability, Less fault proneness and Stability
Replace Constructor with Builder method	Stability

1. “Wrap Return value” refactoring improves testability of the program.
2. “Safe Delete” makes program more adaptable, understandable, less fault proneness and stable.
3. “Replace Constructor with Builder method” makes program more stable.

From Table 2 and Table 5, we found that for other quality attributes inconsistent results are coming where some metrics values are needed to be ignored to improve the quality to certain limit.

1. “Wrap return method” makes the program less fault proneness if increased LOC effect is ignored.
2. “Wrap Return Method” makes system more adaptable when WMC is ignored.

Summing up the analysis part, we concluded that from Table 6 there are few refactoring methods which improve certain quality attributes and hence Hypothesis 1, Hypothesis 3, Hypothesis 5, Hypothesis 7 and Hypothesis 8 are rejected.

From the analysis part of Table 2 “wrap return method value” refactoring changes most of the metric values to desirable state and hence to certain limit improves every quality attribute. Therefore the Hypothesis 2, Hypothesis 4, Hypothesis 6 and Hypothesis 9 are rejected.

6. THREATS TO VALIDITY

There are some limitations to extend the result to general case. There are possible numbers of threats to validity as the few selective classes are taken from the project. The results may vary when implemented on the whole system and when the scenario is changed. We have applied the refactoring on class level not on the system level.

Another possible threat is the correlation between the internal metrics and the external software quality attributes; we have not put validation from our side and directly took the result of previous research.

7. CONCLUSION

Refactoring methods are applied to improve the software quality attribute but the effect of refactoring on particular quality attribute is still ambiguous. In this paper, we applied fourteen refactoring methods and noticed that they effect randomly on different software quality attributes. We classified the refactoring methods which improve a set of metrics which belongs to the attribute, method, coupling, cohesion and inheritance category of software. We focused on different external quality attributes, which are Reusability, Complexity, Maintainability, Testability, Adaptability, Understandability, Fault Proneness, Stability and Completeness and found the effect of refactoring methods on them. By looking at the results, we found that there are few refactoring methods which particularly improve a certain quality attributes of software, which can help the developer to choose them. Our work concludes that refactoring improves the quality of software but developers need to look for the particular refactoring method for desirable quality attribute.

Future research can also test and verify the result on bigger projects and can come up with general relation between refactoring and quality attributes.

ACKNOWLEDGMENT

I would like to acknowledge the support and guidance of my teachers for the excellent guidance throughout the entire work. I am thankful to university resource center for providing the resources for the work.

I would like to thank my parents and friends for all their support during my studies.

REFERENCES

[1] Cinnéide, Mel Ó., Dermot Boyle, and Iman Hemati Moghadam, (2011), "Automated refactoring for testability" , In Software Testing Verification and Validation Workshops (ICSTW), IEEE Fourth International Conference, pp. 437-443, IEEE.

[2] Francisco Zigmund Sokal, Mauricio Finavaro Aniche and Marco Aurelio Gerosa, (2013), “Does The Act Of Refactoring Really Make Code Simpler?, A Preliminary Study”.

[3] Elish, Karim O., and Mohammad Alshayeb. (2009), "Investigating the Effect of Refactoring on Software Testing Effort" In Software Engineering Conference, APSEC'09, Asia-Pacific, pp. 29-34, IEEE.

[4] Bruntink, Magiel, and Arie van Deursen, (2006), "An empirical study into class testability", Journal of systems and software 79, no. 9, pp. 1219-1232.

[5] Kataoka, Y., Imai, T., Andou, H. and Fukaya, T., (2002), "A quantitative evaluation of maintainability enhancement by refactoring", Software Maintenance, Proceedings International Conference, pp.576-585.

[6] Mohammad Alshayeb, (2009), “Empirical Investigation Of Refactoring Effect On Software Quality”, Volume 51, Issue 9, Pages 1319-1326, Elsevier.

[7] M.Fowler, K. Beck, J. Brant, W.Opdyke and D. Roberts, (1999), “Refactoring: Improving the Design of Existing Code”, Addison Wesley.

[8] W.C Wake, (2003), “Refactoring Workbook”, Addison Wesley.

[9] Dandashi Fatma, (2002) "A method for assessing the reusability of object-oriented code using a validated set of automated measurements", In Proceedings of the 2002 ACM symposium on applied computing, pp. 997-1003, ACM.

[10] www.jhotdraw.org.

[11] www.sourceforge.net.
[12] www.jetbrains.com
[13] Opdyke, William F., (1990) "Refactoring: An aid in designing application frameworks and evolving object-oriented systems", In Proc. 1990 Symposium on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA).
[14] IEEE, (1991), Std. 610.12 – IEEE Standard Glossary of Software Engineering Terminology, The Institute of Electrical and Electronics Engineers.
[15] ISO/IEC, (1991), 9126 Standard, Information Technology – Software Product Evaluation – Quality Characteristics and Guidelines for their Use, Switzerland, International Organization for Standardization.
[16] Kayarvizhy, N. and Kanmani, S., (2011) "Analysis of quality of object oriented systems using object oriented metrics," Electronics Computer Technology (ICECT), 3rd International Conference on, vol.5, no., pp.203-206.
[17] Chidamber, S.R and Kemerer, C.F., (1994) “A metrics suite for object oriented design," Software Engineering, IEEE Transaction, vol.20, no.6, pp.476-493.
[18] Www. Refactoring.com.
[19] Li, W and Henry, S., (1993) "Maintenance metrics for the object oriented paradigm," Software Metrics Symposium, Proceedings, First International, pp.52-60.
[20] Daniel Rodriguez and Rachel Harrison, (2001),“An Overview of Object-Oriented Design Metrics”.
[21] Khalid, Sadaf, Saima Zehra and Fahim Arif, (2010) "Analysis of object oriented complexity and testability using object oriented design metrics", In Proceedings of the 2010 National Software Engineering Conference, ACM.
[22] Srivastava, Sandeep, and Ram Kumar, (2013) "Indirect method to measure software quality using CK-OO suite." In Intelligent Systems and Signal Processing (ISSP), 2013 International Conference on, pp. 47-51, IEEE.
[23] Elish, Mahmoud O. and David Rine, (2003) "Investigation of metrics for object-oriented design logical stability", In Software Maintenance and Reengineering Proceedings, Seventh European Conference on, pp. 193-200, IEEE.
[24] Basili, Victor R., Lionel C. Briand and Walcélio L. Melo, (1996) "A validation of object-oriented design metrics as quality indicators", Software Engineering, IEEE Transactions on 22, no. 10, pp. 751-761.
[25] Jehad Al Dallal, (2013) "Object-oriented class maintainability prediction using internal quality attributes", Information and Software Technology 55, no. 11.
[26] Subramanian, Nary, and Lawrence Chung, (2001) "Metrics for software adaptability", Proc. Software Quality Management (SQM 2001).
[27] Abreu, Fernando B, (1995) "The MOOD Metrics Set," Proc. ECOOP'95 Workshop on Metrics.
[28] Stroggylos, Konstantinos, and Diomidis Spinellis., (2007) "Refactoring--Does It Improve Software Quality?” proceedings of the 5th International Workshop on Software Quality, IEEE Computer Society.
[29] Vasudeva Shrivastava, S.,and V. Shrivastava. (2008) "Impact of metrics based refactoring on the software quality: a case study". TENCON 2008 IEEE Region 10 Conference, IEEE.
[30] Sharma, Tushar., (2012), "Quantifying Quality of Software Design to Measure the Impact of Refactoring”. Computer Software and Applications Conference Workshops, IEEE 36th Annual.