Analysis of Code Refactoring Impact on Software Quality

Amandeep Kaur[1] and Manpreet Kaur[2]

[1]*Computer Science and Engg. Department, Punjab Technical University,Jalandhar, India, amansidhu1092@gmail.com*
[2]*Computer Science and Engg. Department, Punjab Technical University,Jalandhar,India, manpreet.kaur@bbsbec.ac.in*

Abstract. Code refactoring is a "Technique used for restructuring an existing source code, improving its internal structure without changing its external behaviour". It is the process of changing a source code in such a way that it does not alter the external behaviour of the code yet improves its internal structure. It is a way to clean up code that minimizes the chances of introducing bugs. Refactoring is a change made to the internal structure of a software component to make it easier to understand and cheaper to modify, without changing the observable behaviour of that software component. Bad smells indicate that there is something wrong in the code that have to refactor. There are different tools that are available to identify and remove these bad smells. It is a technique that change our source code in a more readable and maintainable form by removing the bad smells from the code. Refactoring is used to improve the quality of software by reducing the complexity. In this paper bad smells are found and perform the refactoring based on these bad smell and then find the complexity of program and compare with initial complexity. This paper shows that when refactoring is performed the complexity of software decrease and easily understandable.

1 Introduction

1.1 Refactoring

Refactoring is a change made to the internal structure of a software component to make it easier to understand and cheaper to modify, without changing the observable behaviour of that software component [1]. This definition is attractive because it not only defines what refactoring is, but also why refactoring are performed. Refactoring are performed to improve the functional factorings of a software system in order to increase understand ability and modifiability.

Refactoring is the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure. It is a way to clean up code that minimizes the chances of introducing bugs.

Code refactoring is a "disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behaviour", undertaken in order to improve some of the non-functional attributes of the software. Typically, this is done by applying a series of "Refactoring", each of which is a (usually) tiny change in a computer program's source code that does not modify its conformance to functional requirements. Advantages include improved code readability and reduced complexity to improve the maintainability of the source code, as well as a more expressive internal architecture or object model to improve extensibility.

Refactoring is the process of taking existing code and changing it in some way to make it more readable and perhaps less complex. The key thing to note, is that when the code is changed, is does not affect the underlying functionality of the code itself. So the changes you're making are literally just to make the code easier.

Manual refactoring are often error-prone and time-consuming [2]. For instance, renaming a method requires checking that the method's new name is not yet in use as well as updating all invocations. Besides being obviously time-consuming this operation is also error-prone because polymorphism may cause a forgotten update to compile correctly but change the software's behaviour inadvertently. This requires the maintainer to manually locate the changed functionality and update the omitted invocation. As a result, these refactoring are often not performed and the software's structure deteriorates as a result of functional changes.

Automated refactoring tools can reduce these problems. If performed by a tool that can guarantee that the refactoring it performs are behaviour-preserving, the error-prone aspect is mitigated. Furthermore, performing the required analysis automatically can drastically reduce the time required to perform refactoring. Automated refactoring is made possible through the use of preconditions that if satisfied guarantee that a refactoring is behaviour-preserving [3]. Also, composing larger

Amandeep Kaur: amansidhu1092@gmail.com

refactoring from smaller ones means that a relatively small set of automated refactoring can be used to perform a large set of refactoring.

When performed by a tool, refactoring consists of at least two steps [2]. The first is analysis, where the program to be refactored is analyzed in order to determine whether the desired refactoring's preconditions are satisfied. If this is the case, the second step is executed, the actual transformation of the program source code. Both steps must take both the syntax and semantics of the programming language the tool supports into consideration, making it a considerable effort to implement a refactoring tool from scratch. This may explain why refactoring tools are often integrated with other development tools such as IDEs, since these typically expose a large part of this required functionality which can be reused by the refactoring tool. No matter how complex a refactoring tool is, in theory it only needs to be implemented once for each programming language where automated refactoring functionality is desired and then evolved along with the language it supports. Given the large developer communities and companies backing the most popular programming languages, several refactoring tools are readily available.

1.2 Bad smell

Bad smells indicate that there is something wrong in the code that we have to refactor. Bad smells are design flaws in the code. There are many tools that are available to identify the bad smells and remove these bad smells by using refactoring tools and by using refactoring technique. Refactoring is a technique that restrict our source code in a more readable and maintainable form by removing the bad smells from the code. Refactoring does not change the external behaviour of software[7].

Bad smells are potential problems that can be remove through refactoring. There are various kind of bad smells that make our source code difficult to understand and modify. Bad in a code is not any problem but may lead to any mistake in future. We can detect and refactor this bad smells through various tools. Jdeodorant are such kind of tools that are used to detect the bad smells [7]. Bad smells are of following types:

- LARGE CLASS- means a class that is too large. Size of the large class is too much large. This type of class is difficult to understand and it is too much hard to understand that which functioning is performed by this class.
- LONG METHOD- is a method that is too long. Long methods are same as large classes. Long methods also leads to confusion for the new developer and these are also very much difficult to understand.
- DUPLICATE CODE- is a code that is repeated at too many places in a same source code. The problem arises when we try to update this code. We have updated the duplicate code at all the places in which this code is placed. If we forget to update the code on single place, it may create problem. Hence duplicate code is difficult to maintain.
- FEATURE ENVY - is a bad smell that violates the principle of its class in a source code. From many discussion we found that this a bad smell that is not interested to use its own source class but interested to use any another source class.

All these bad smells can be clean up by using the refactoring.

1.3 Refactoring Techniques

Refactoring Technique is used to remove the bad smells from code and make code clean. Refactoring is a set of techniques, procedures and steps to keep your source code as clean as possible. Clean code, on the other hand, describes how well-written code should look in ideal conditions. In a pragmatic sense, all refactoring represents simple steps toward clean code. There are some basic techniques that are used for refactoring[7]-

- EXTRACT METHOD:- Extract method means extract a method that appear at many places in the source code and place it in a different method.
- INLINE METHOD:-Its working is totally opposite to the extract method. A method of body is as clear as its name. Put the body of the method into the body of its caller. Then remove the method.
- MOVE METHOD:-Move method means moving the method from one class to another when a class has too many functions to do.
- INLINE CLASS:-This method is applicable to those classes which are not doing too much work. With the help of inline class we can move the functionality of this class to another class and remove the class.
- RENAME METHOD:-Rename method simply means renames any method. We can rename the method according to the functionality of the method. It makes our code easier to understand.
- REPLACE ARRAY WITH OBJECT:-If we have many or different elements. We can replace this array with an object, in this object each element have specific field.

Refactoring technique is used to make code clean . After finding the bad smell in code then we apply the respected refactoring to remove the bad smell in code. After performing the refactoring the code complexity is decrease and it is easily understandable by anyone.

1.4 Refactoring Tool

ECLIPSE:- Eclipse is an IDE(Integrated Development Environment). The Eclipse platform which provides the foundation for the Eclipse IDE is composed of plug-ins and is designed to be extensible using additional plug-ins. Developed using Java, the Eclipse platform can be used to develop rich client applications, integrated development environments and other tools. Eclipse can be used as an IDE for any programming language for which a plug-in is available. In it there is a workspace and an extensible plug-in for customizing the environment. Written in java language it is used to develop application. Eclipse may also helpful to create the application in C, C++, PHP and COBOL and many more languages. Eclipse also provide an option to refactor the source code. There are many plug-in for eclipse that can used to detect the bad smells in the code. To refactor the bad smells it provides various refactoring techniques.

In order to increase the availability of refactoring tools for DSLs it is worthwhile to investigate whether refactoring tool implementation can be simplified in order to be able to develop refactoring tools (for DSLs) faster or even generate them. To this end, it is useful to examine the implementation of existing refactoring tools in order to gain insight into the complexities of their development. Eclipse is a suitable candidate for such an examination, since it is a very widely used IDE and it supports a large set of refactoring out-of-the-box, especially in the JDT, the Java Development Tooling plug-in[18].

Plugins-

JDeodorant is an Eclipse plugin that are used to detect the bad smells. JDeodorant was simplistic in both design and usage. It support 4 type of bad smells-God class, Long method, Type checking, Feature envy. JDeodorant that automatically identifies the Feature Envy, God Class, Long Method and Switch Statement (in its Type Checking variant) code smells in Java programs. This plugin is used to find the bad smells in the software.

EMF Refactor are such kind of tool that are used to perform the refactoring on code. It support various type of refactoring like Rename Method/Class, Extract Method/Class, Inline Method , Move ,Replace etc. This plugin is used to remove the bad smells in the software.

Eclipse Metrics Plugin are such kind of tool that are used to calculate the various type of metrics in the program. It calculate Lines of code, No. of attributes, No. of Classes, No. of Methods, Cohesion, Coupling, McCabe Cyclomatic Complexity, Weighted methods per class. This plugin is used to find the complexity of software to detect the quality of software.

2 Literature Survey

Fowler et al., [1] say that, Refactoring is basically the object-oriented variant of restructuring: “the process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure”. They outlines four different occasions when a programmer should refactor. First, when code is duplicated for the second time, a programmer should factor out the duplication. Second, they should refactor when functionality needs to be added, but the existing code is hard to understand or the addition is not easy to make because of the existing design. Third, they should refactor when a bug needs to be fixed and refactoring the code will help make the code clearer and expose the bug. Finally, the should refactor when programmers are doing a code review and refactoring will immediately produce code that everyone understands.

Martin Fowler[1] discusses in his paper that how refactoring improve the existing design. He introduce deeply about refactoring in his paper. This paper also discuss more about the bad smells. How to detect these smells and tells that which refactoring technique is applicable to remove this bad smell. Martin also introduce that a code that have bad smells are hard to maintain and hard to modify. A bad smell is an indication of some problem in the code, which requires refactoring to deal with. Many tools are available here for detection and refactoring of these code smells. These tools vary greatly in detection methodologies and acquire different competencies. In this work, we studied different code smell detection tools minutely and try to comprehend our analysis by forming a comparative of their features and working scenario. We also extracted some suggestions on the bases of variations found in the results of both detection tools. This helps us to select the tool to refactor the code.

Sandeep Kaur[7], says that, Bad smells indicate that there is something wrong in the code that we have to refactor. Bad smells are design flaws in the code. There are many tools that are available to identify the bad smells and remove these bad smells by using refactoring tools and by using refactoring technique. Refactoring is a technique that restrict our source code in a more readable and maintainable form by removing the bad smells from the code. Refactoring does not change the external behaviour of software. In this paper we discussed about tools and techniques to refactor the source code.

Roberts [2] says that, Refactoring consists of at least two steps, The first is analysis, where the program to be refactored is analyzed in order to determine whether the desired refactoring’s preconditions are satisfied. If this is the case, the second step is executed, the actual transformation of the program source code. Both steps must take both the syntax and semantics of the programming language the tool supports into consideration, making it a considerable effort to implement a refactoring tool from scratch. This may explain why refactoring tools are often integrated with other development tools such as IDEs, since these typically expose a large part of this required functionality which can be reused by the refactoring tool.

Mealy and Strooper [4] say that, Refactoring is a method which used to rearrange and modify the existing code in a way that the intentional behaviour of code stays the same. Refactoring allows to simplify and improve both performance and readability of your code. One of the key issue in software refactoring is source code that should be refactored. It should be implemented by the object oriented programs. Kent beck and Martin fowler calls them Bad smells, indicating that some part of the source code are terrible. Sometimes in other words bad smells are assigned as the duplicate code. Duplicate code is a computer program sequence of source code that occurs more than once. Improving the design that often includes removing duplicate code .

Roberts D. Brant [2] says that refactoring is provided as a program transformation that has a precondition and a post-condition that a program must satisfy for the refactoring to be easily applied. Each program is thought to have a specification for it and that specification is satisfied (or unsatisfied) by a test suite. A refactoring is therefore behaviour preserving if it satisfies the original test suite. If a new component is added to the program, the program must satisfy the original test suite plus any additional tests. When satisfying the original test suite, one must recognize that this is the conceptual original test suite that is satisfied.

Kumar and Chanaky[20] say that A, Code and design smells are the indicators of potential problems in code. They may obstruct the development of a system by creating difficulties for developers to fulfil the changes. Detecting and resolving code smells, is time-consuming process. Many number of code smells have been identified and the sequences through which the detection and resolution of code smells are operated rarely because developers do not know how to rectify the sequences of code smells. Refactoring tools are used to facilitate software refactoring and helps the developers to restructure the code. Refactoring tools are passive and used for code smell detection. Few refactoring tools might result in poor software quality and delayed refactoring may lead to higher refactoring cost. A Refactoring Framework is proposed which instantly detects the code smells and changes in the source code are analyzed by running a monitor at the background. The proposed framework is evaluated on different non trivial open source applications and the evaluation results suggest that the refactoring framework would help to avoid code smells and average life span of resolved smells can be reduced.

Elish [22] says that, Refactoring to patterns allows software designers to safely move their designs towards specific design patterns by applying multiple low-level refactorings. There are many different refactoring to pattern techniques, each with a particular purpose and a varying effect on software quality attributes. Thus far, software designers do not have a clear means to choose refactoring to pattern techniques to improve certain quality attributes. This paper takes the first step towards a classification of refactoring to pattern techniques based on their measurable effect on software quality attributes. This classification helps software designers in selecting the appropriate refactoring to pattern techniques that will improve the quality of their design based on their design objectives. It also enables them to predict the quality drift caused by using specific refactoring to pattern techniques.

3 Problem Formulation

Producing software is a very complex process that takes a considerable time to evolve. Poorly designed software systems are difficult to understand and maintain. Software maintenance can take up to 50% of the overall development costs of producing software. One of the main attributes to these high costs is poorly designed code, which makes it difficult for developers to understand the system even before considering implementing new code. In the context of software engineering process, Software Refactoring has a direct influence on reducing the cost of software maintenance through changing the internal structure of the code, without changing it external behaviour. Refactoring is a technique for restructuring an existing body of code.

- Code is not easily maintainable and difficult to understand.
- Design of program is more complex and difficult to find bugs in the program.
- Code review is more time consuming.

To remove these problems in our java program we perform refactoring using Eclipse Tool. After performing refactoring, the internal structure of program is modified and its external behaviour remains same.

Modifications and Improvements

- After refactoring, code is easy to understand.
- Refactored code is easily maintainable and reduce the code maintenance cost.
- Refactored code is of better quality and reduce the chance of introducing bugs.

Objectives

- To find the code smells.
- To remove bad smells in code.
- To improve the quality of code.
- To improve code readability and reduced complexity.
- To make software easy to understand.
- To reduce the maintenance cost.

4 Research Methodology

This section gives description of step involved for Refactoring-

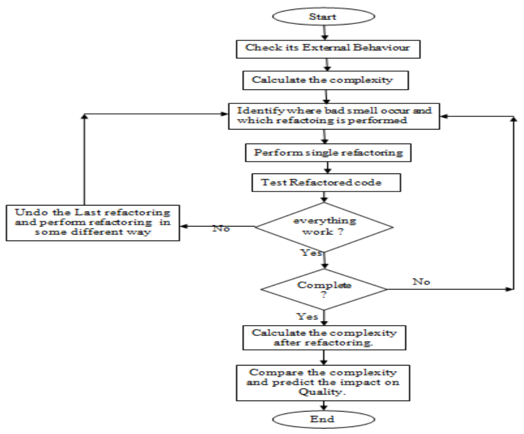
1. Run the program to check its external behaviour and ensure that it is unchanged after refactoring.
2. Before applying any single refactoring, calculate the complexity of program.
3. Identify where the code should be refactored- Determine which refactoring should be applied to the identified places.

4. Make a small change- a single refactoring without changing the outer behaviour of the code.

5. Test Refactored code if everything work's, move on to the next refactoring.

6. If fails rollback the last smaller change and repeat the refactoring in a different way.

7. After applying all the refactoring technique, Calculate the complexity to determine the impact of refactoring on Quality.


Start
Check its External Behaviour
Calculate the complexity
Identify where bad smell occur and which refactoing is performed
Perform single refactoring
Test Refactored code
Undo the Last refactoring and perform refactoring in some different way
No
everything work ?
Yes
Complete ?
No
Yes
Calculate the complexity after refactoring.
Compare the complexity and predict the impact on Quality.
End

5 Experiment

The experiment is done by using the eclipse tool and jdeodrant plugin as bad smell detection tool and eclipse metrics plugin as to calculate the complexity of source code. We take an source code of library software which is written in Java.

Bad smells	**Jdeodorant**	**Refactoring technique that can be applied**
Feature envy	Yes	Move method
Type checking	Yes	Replace replace type code with state
Large class	Yes	Move Method
Long Method	Yes	Extract Method

Comparison between Before and After Refactoring

Before refactoring, we can find two types of bad smells in our project and the initial complexity is calculate. When bad smells find in our project, then we remove these bad smells by applying the appropriate techniques of refactoring. After applying all type of refactoring then again we calculate the complexity and compare it with initial complexity and check the difference between these two. Complexity can also be measure in the form of McCabe Cyclomatic Complexity, Weighted Methods per class(WMC), Lack of Cohesion of Methods(LCOM), Depth of Inheritance Tree(DIT). A low number for DIT and WMC implies less complexity and a high number for DIT and WMC implies higher complexity with a higher probability of errors in the code. If LCOM value is small then there is more cohesion between the methods and if the value is large then there is lack of cohesion between the methods. The complexity of our project is reduced after apply the refactoring technique. We can say that refactoring improve the internal structure of our project by decreasing the complexity and improve the quality of our project.

Measures		**Before Refactoring**	**After Refactoring**
Bad Smells	Long Method	yes	No
	Feature Envy	No	No
	Long Class	Yes	No
	Type Checking	No	No
Comple -xity	McCabe Cyclomatic Complexity	3.7	2.701(↓)
WMC	Weighted Methods Per Class(WMC)	13.45	12.25(↓)
LCOM	Lack of Cohesion of Methods(LCOM)	0.43	0.397(↓)
DIT	Depth of Inheritance Tree	3.864	3.25(↓)

6 Conclusion

Refactoring is an important and easy activity to refactor the source code in a well manner. Refactoring makes a code easier to understand and improve the quality of code. We take a source code of library software which is written in Java. We can detect bad smells by JDeodorant and find the complexity by Metrics plugin. Bad smells make our source code more difficult to manage. By applying refactoring using Eclipse, we can refactor these bad smells and make our source code essay to understand. After applying the refactoring, calculate the complexity of project and compare it with initial complexity and then check the result. The complexity of our project is reduced after apply the refactoring and improve the quality of our project. Also we can say that refactoring reduces the maintenance cost because the complexity of software is decrease and it is easily understandable.

References

1. Martin Fowler, Kent, John Brant, William Opdyke, Don Roberts, D. B. "Refactoring: Improving the Design of Existing Code", Addison-Wesley, New York, (1999).

2. Roberts, D. B “Practical Analysis for Refactoring”, PhD thesis, Department of Computer Science , University of Illinois at Urbana-Champaign, (1999).
3. Opdyke, W. F , “Refactoring Object-Oriented Frameworks”, PhD thesis, University of Illinois at Urbana-Champaign,(1992).
4. E.Mealy and P.Strooper,”Evaluating software Refactoring Tool support”, Proceeding of Australian Software Engineering Conference, pp. 331-340, (2006).
5. E.Mealy, D.Carrington, P.Strooper, and P.Wyeth, ”Improving usability of software refactoring tools”, Proceeding of Australian Software Engineering Conference , pp.307-318, (Apr.2007).
6. Tom Mens and Tom Touwe, “A survey of software refactoring” IEEE Transactions on software Engineering ,vol.30, no.2, pp. 126-139, (Feb 2004).
7. Sandeep kaur , “ Review on Identification and Refactoring of Bad Smells using Eclipse”, International Journal For Technological Research In Engineering (IJTRE) Volume 2, (March-2015).
8. R. Fanta and V. Rajlich, “Reengineering object-oriented code,” in Proceeding of International Conference on Software Maintenance, pp. 238–246, 1998, IEEE Computer Society.
9. Eclipse Modeling Framework (EMF) - http://www.eclipse.org/emf/
10. EMF Refactor - http://www.eclipse.org/emf/refactor/.
11. JDeodorant - https://marketplace.eclipse.org/content/jdeodorant
12. EMF Metrics Plugin - http://sourceforge.net/projects/metrics/
13. Mika V. Mäntylä, Jari Vanhanen, and Casper Lassenius,” A taxonomy and an initial empirical study of bad smells in code”, In Proceedings of International Conference on Software Maintenance (ICSM 2003), IEEE Computer Society pages 381–384, Amsterdam, The Netherlands, (September 2003).
14. Nikolaos Tsantalis and Alexander Chatzigeorgiou, “Identification of move method refactoring opportunities”, IEEE Transactions on Software Engineering, 35(3):347–367, (2009).
15. Nikolaos Tsantalis, “Identification Of Move Method Refactoring Opportunities”, IEEE Transactions On Software Engineering, Vol. 35, No. 3, (May/June 2009).
16. S.H. Kannangara, “An Empirical Evaluation Of Impact Of Refactoring On Internal And External Measures Of Code Quality”, International Journal Of Software Engineering & Applications (Ijsea), Vol.6, No.1, (January 2015).
17. J. van den Bos, “Refactoring (in) Eclipse”, Master Software Engineering, Universiteit van Amsterdam, Master’s thesis , (August 2008).
18. Mesfin Abebe and Cheol-Jung Yoo, “ Trends, Opportunities and Challenges of Software Refactoring: A Systematic Literature Review”, International Journal of Software Engineering and Its Applications Vol.8, No.6 ,pp.299-318,(2014).
19. Emerson Murphy-Hill and Andrew P. Black, “Refactoring Tools: Fitness for Purpose”, Department of Computer Science, Portland State University Portland, Oregon, (May 7, 2008).
20. D. Raj Kumar, G.M. Chanakya, “Refactoring Framework for Instance Code Smell Detection”, International Journal of Advanced Research in Computer Engineering & Technology (IJARCET) Volume 3 Issue 9, (September 2014).
21. Emerson Murphy-Hill, Chris Parnin, And Andrew P. Black, “How We Refactor, And How We Know It”, IEEE Transactions On Software Engineering, Vol. 38, No. 1, (January/February 2012).
22. Karim O. Elish, “Using Software Quality Attributes to Classify Refactoring to Patterns”, Journal Of Software, Vol. 7, No. 2, (February 2012).