International Journal on Future Revolution in Computer Science & Communication Engineering ISSN: 2454-4248
Volume: 4 Issue: 2 232 – 235

Barriers to Refactoring: Issues and Solutions

Zeba Khanam,
College of Computing and Informatics,
Saudi Electronic University,
KSA

Abstract:- Refactoring mechanism is commonly used in software development. Though Object oriented programming promotes ease in designing reusable software but the long coded methods makes it unreadable and enhances the complexity of the methods. The common code defects are associated with large classes and methods. To ease up the code comprehension Extract method, Extract class serves as a comfortable option to reduce the disorganization and duplication of the code to produce more fine grained methods and classes. Though refactoring serves as an important mechanism to improve the software quality whether performed manually or in an automated way with the help of tools or IDEs but there are umpteen cases where refactoring could lead to deterrent effect. This paper intends to explore the various problems and barriers associated with refactoring and specifically while extracting the code (extract method, extract class, extract interface) and their solutions.

*****

1. Introduction

The decision to refactor may not be difficult but to refactor in the best possible way without erroneous manipulation of the code with some known benefits is what is more symbolic . While extraction of code fragments looks like a simple computation and well supported by IDEs such as Eclipse, intelliJ IDEA etc, however identification of code for extraction can sometimes be an arduous task. Sometimes the code extraction whether it might be to a method or a class varies with different programming paradigms and languages aswell. For example the mechanism in a completely procedural language differs from OOP whereas the extraction methodology in Aspect oriented programming could be a completely different phenomena, the Component Based Development on the other hand engages in extraction of reusable components[10][11][15]. The problem that consumes most of the time is in picking the most appropriate refactoring candidate. The research [1] depicts the problem of finding the most suitable refactoring candidate for long methods written in Java. The approach adopted investigates for the most appropriate refactoring candidates and ranks them using a scoring function that aims to improve readability and reduce code complexity. The mechanism applied is length and nesting reduction as complexity indicators. Many researchers have worked on the Extract method refactoring [1][2][12][13], the work in [1] generates a static analysis tool to find extract method opportunities based on variable references.This tool creates an intermediate representation of the method for the tree-map visualization tool.

Extract Method is generally applied in combination with other core refactorings such as Move Method and Extract Class [7]. These refactoring are usually done after the extract method is already applied. This paper primarily focuses on problems associated with code extraction.

Refactoring techniques tend to impact different quality aspects of software programs such as cohesion, complexity and readability. The extract method refactoring has been dealt in various ways such as extracting fragments to aspects using Aspect oriented programming, identifying the refactoring opportunities automatically, or extraction of the code chunks as a separate method that collaborate to provide a specific functionality.[2][3][5][6][15]

Most of the researchers are of the view that proper test suites could be the best solution and should be in place, but the author [8] states that semantic preconditions checking could be an effective solution. A recent study [16] investigated the reasons effecting the developer's refactoring decision whether its driven due to design related issues or any other. However, the results prove that the decision to refactor or not are not dependent on design considerations.

2. Issues and Solutions Associated with Refactoring

The gains associated with refactoring are usually in making the code look simpler and with a better design that would be easier for the other developers to comprehend. Whether the refactoring is performed manually or in an automated way they tend to have some shortcomings associated with some of them that might be exposed immediately or in the long run. In this section we intend to highlight a few problems with a few of well known commonly performed refactorings and their likely solutions.

2.1 Not using a Refactoring Tool

Very often the most common cause of refactoring defect is due to its manual implementation and not the code, due to the unavailability of appropriate tool. The refactoring like changing method signatures, renaming variables and methods, moving variables between the classes etc may lead to unidentified bugs and harm the entire code eventually.

For example, changing the name of a local variable may sound a simple task, but if there is an existent class level variable with the same name then the new name would hide it. Refactoring tools on the other hand take all these warnings into account, while doing it manually maybe misleading. Therefore the solution is to initiate your refactorings with a suitable tool.

2.2 Refactoring not accompanied with unit tests

Unit testing forms a necessary practice of the refactoring process and if not performed after refactoring may lead to the obvious danger of introducing undetected defects in the refactored code. Thus validating for changes is a difficult task. However, another consideration that the developers may usually be casual about is that the legacy code could already have an undetected error that is not known until the refactoring is done. The developer's response would be to assume that the changes introduced in the code, might have introduced the bug.

That would lead to a search through the latest changes made to find the error, though the actual problem lies in the code that remains untouched.

So the best option is to first create a set of unit tests for the existing code so that if there are any defects prior to the refactoring they may get detected before incorporation of any change. The test driven development that is gaining momentum too is based on the same principal [6] that whenever a new addition is done to the code, the developers have to design the test cases for the requirements and the code is supposed to pass it and refactored for a better design. Sometimes TDD is used to improve the design or debug the legacy code.

2.3 Refactoring the code bound with many external interfaces

The code that interacts with many external interfaces can be complicated and error prone for refactoring, especially for loosely typed or string parameters. For example refactoring the system can lead to many changes and may accidentally introduce some defects. A particular data that was unexpected by the receiver system when sent by the external system had been working earlier but as was unexpected by the receiver may not work after the changes. These errors are hard to detect and finding them during the development cycle is almost not possible. They can be detected only during a full QA cycle or during the production process.

The optimal solution for this kind of problem is to record all the data sets transmitted between the internal code and the external interfaces for a specified duration. Then perform refactoring of the system and design tests for the system using the captured data. The test results would establish if the system is capable of handling the live traffic correctly or not.

2.4 Change method signature

The change method signature refactoring deals with changing the method name, changing the parameter names or the return types, adding or removing a new parameter, or reordering the parameters, changing the visibility scope etc.

But the manipulation of method signatures can lead to code defects not identifiable easily, especially for methods where all the parameters are of the same time. For example a method that has two strings as parameters, but due to some logical reason the developer changes the order of the parameters, now since both are of the same type this situation may create ambiguity because any other developer cannot realize the change and by any chance if the developer forgets to change the order in the method calls, the system won't project it as an error.

Therefore to avoid any ambiguity its best to be meticulous in it and method signature manipulation should be preferably automated to avoid any such risk.

3. One of the currently suggested solutions to ensure safer refactorings is to encode preconditions in dynamic analysis [8]. Though static analysis can effectively check syntactic preconditions, but the semantic preconditions checking is a comparatively complex task in languages like Java where static analysis is hardly able to analyze what executable path will be followed.

In experiment settings it has been shown that developers can have a low understanding of the implications of a refactoring and the preconditions posed on the source code, and that warnings or previews can sometimes be ignored [9], leading to unsafe refactorings.

2.5 Problems while extracting interfaces and superclasses

Improving the code while refactoring often involves generating new interfaces to the class but the manual procedure is quite tiresome as writing the method signatures and other properties could be common cause of errors and code defects, an appropriate automation in this regard is available. But it would be a great idea if the extracted interface would scan through the other classes and prompt the user about the classes that would likely implement the same interface, it would save on a lot of manual effort.

Another concern is while extracting a super class from a very useful class by creating a base class by extracting the most common code. But in Java sometimes the problem may arise that though we have created a base class from this very useful class, we want this class to be again reused in another base class but that won't be possible as the class is already having a parent class and multiple inheritance is not allowed in Java.

2.6 Language independent Refactorings

State of the art suggests that the majority of the refactorings proposed are paradigm or language dependent. But there are a variety softwares that constitute the code belonging two different paradigms such as the AspectJ code embedded in Java [17][20].There are refactorings performed to modify the code from OOP to AOP or from procedural language to AOP(such as C to AspectC or Objective C). Therefore an important consideration with the tool enabled refactorings would be to support cross paradigm refactorings. The research done in this area[18][19] works on implementing the refactorings that are not specific to a language or paradigm and can be applied without bothering about the compatibility.

2.7 Refactoring for Design Problems may not be a very good option

Refactoring may be of good use in improving the design of the code but to naively believe that it can completely change the design and deep rooted mistakes and would produce an upfront design would be an over assumption. Since recognizing the defects and gaps take a long time in the real world as by the time one starts to refactor the actual smell a lot of defects have already piled after application of various patches and continuous corrections, the code evolves into a different shapes. Thats probably when the developer realizes that the software needs probably a different architecture or different sets of abstraction.

Though it may not always be a bad idea as a lot exploration and research has been made on problems in software, wrong applications of designing principles and patterns [14]. Around 25 structural design smells, their roles and their respective refactorings are discussed.

Conclusion

As a developer involved in striving to write the best code, usage of refactoring as a tool for improvement of the code is a common phenomena. As a curious developer one may wish to experiment with the different refactorings available in the popular catalogs of Fowler and Kerievsky.Though the catalogs are quite precise in descriptions but given the varied nature of software with different paradigms and languages the catalogs or the existing information sometimes falls short many a times n describing the refactorings as per the requirements. When practically implementing few of these refactorings the developers usually experience a lot of ambiguity: as the tools available may differ on implementation details or precondition checks that makes it more time consuming and sometimes leads to futile attempt. So this paper has highlighted a few of the popular or common issues that are associated with various refactorings and proposes the technique to be adopted in such cases.

Also the tools that are used for the purpose of refactoring don't generally have any option of assessing the refactorings and predicting their likely outcome or result on the refactored code.If this property somehow could be added in a tool a lot of ambiguity surrounding when and how to refactor the code could be solved.

References

[1] Kaya, Mehmet, "Identifying Extract Class and Extract Method Refactoring Opportunities Through Analysis of Variable Declarations and Uses" (2014). Dissertations - ALL. Paper 53.

[2] Nikolaos Tsantalis and Alexander Chatzigeorgiou , "Identification of extract method refactoring opportunities for the decomposition of methods", Journal of Systems and SoftwareVolume 84, Issue 10, October 2011, Pages 1757-1782

[3] Sofia Charalampidou; Apostolos Ampatzoglou; Alexander Chatzigeorgiou; Antonios Gkortzis; Paris Avgeriou, "Identifying Extract Method Refactoring Opportunities Based on Functional Relevance", IEEE Transactions on Software Engineering (Volume: 43, Issue: 10, Oct. 1 2017)

[4] Roman Haas1 and Benjamin Hummel,Deriving Extract Method Refactoring Suggestions for Long Methods. Software Quality Days 2016 (SQD'16), 2016.

[5] Rizvi. S and Khanam Z "Assessment of the Impact of Aspect Oriented Programming on Refactoring Procedural Software". Computers and Mathematics in Automation and Materials Science, MATHIMA24,Cambridge,USA.2014

[6] Zeba Khanam and Mohammed Najeeb Ahsan, "Evaluating the Effectiveness of Test Driven Development: Advantages and Pitfalls"(2017). International Journal of Applied Engineering Research ISSN 0973-4562 Volume 12, Number 18 (2017) pp. 7705-7716

[7] Nikolaos Tsantalis and Alexander Chatzigeorgiou, "Identification of Extract Method Refactoring Opportunities" in 13th European Conference on Software Maintenance and Reengineering, 2009. CSMR '09.

[8] Anna Maria Eilertsen,"Making Software Refactorings Safer",Master Thesis,June ,2016.

[9] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson. Use, disuse, and misuse of automated refactorings. In 34th International Conference on Software Engineering (ICSE 2012), pages 233–243. IEEE, 2012.

[10] Rizvi, S., Khanam, Z. "A methodology for refactoring legacy code." In: International Conference on Electronics

Computer Technology (ICECT 2011), pp. 198–200 (2011), IEEE Xplore.

[11] Hironori Washizaki and YoshiakiFukazawa, "A technique for automatic component extraction from object-oriented programs by refactoring", Elsevier,Science of Computer Programming, Volume 56, Issues 1–2, April 2005, Pages 99-116

[12] D. Silva, R. Terra, M. T. Valente, "Recommending automated extract method refactorings", *Proceedings of the 22nd International Conference on Program Comprehension (ICPC'14)*, pp. 146-156, 2014.

[13] Sihan Xu, Aishwarya Sivaraman, Siau-Cheng Khoo, Jing Xu:
GEMS: An Extract Method Refactoring Recommender. 24-34, 28th IEEE International Symposium on Software Reliability Engineering, ISSRE 2017, Toulouse, France, October 23-26, 2017. IEEE Computer Society 2017, ISBN 978-1-5386-0941-5.

[14] Girish Suryanarayana, Ganesh Samarthyam and Tushar Sharma," *Refactoring for Software Design Smells. Managing Technical Debt", ISBN: 978-0-12-801397-7*

[15] Z Khanam, SAM Rizvi. "Aspectual Analysis of Legacy Systems: Code Smells and Transformations in C", International Journal of Modern Education and Computer Science, Volume 5, Issue 11, Page 57, 2013.

[16] Ewan Tempero, Tony Gorschek, Lefteris Angelis , "Barriers to Refactoring". Communications of the ACM, Vol. 60 No. 10, Pages 54-61
10.1145/3131873",2017

[17] SAM Rizvi, Z Khanam . "A Comparative Study of using Object oriented approach and Aspect oriented approach for the Evolution of Legacy System", International Journal of Computer Applications 1 (782), 0975-8887, 2010.

[18] Harun, M.F. and Lichter, H. Towards a technical debt-management framework based on cost-benefit analysis. In *Proceedings of the 10th International Conference on Software Engineering Advances* (Barcelona, Spain). International Academy, Research, and Industry Association, 2015.

[19] Sérgio Bryton, Fernando Brito e Abreu," Modularity-Oriented Refactoring",2007

[20] Z Khanam, SAM Rizvi, "Assessment of the Impact of Aspect Oriented Programming on Refactoring Procedural Software", Computers and Mathematics in Automation and Materials Science, MATHIMA-24, Cambridge, USA, 2014.