Innovative Systems Design and Engineering www.iiste.org
ISSN 2222-1727 (Paper) ISSN 2222-2871 (Online)
Vol 2, No 4, 2011

Refactoring for Multi-Dimensional Reusability

Bashir Ahmad
Institute of Computing and Information Technology
Gomal University, PAKISTAN
bashahmad@gmail.com

Shakeel Ahmad
Institute of Computing and Information Technology
Gomal University, PAKISTAN
Shakeel_1965@yahoo.com

Sheikh Muhammad Saqib
Institute of Computing and Information Technology
Gomal University, PAKISTAN
saqibsheikh4@hotmail.com

Muhammad Zubair Asghar
Institute of Computing and Information Technology
Gomal University, PAKISTAN
Zubair_icit@yahoo.com

Muhammad Ahmad Jan
Institute of Computing and Information Technology
Gomal University, PAKISTAN
mr_ahmadjan@yahoo.com

Abstract

Source code should be simpler, easy to read and easy to understand. This slogan is not only relates to change the existing code for current service, but also has an association with reusability. Refactoring is a best idea for above issues i.e. keeping the code simple and support the emergent design practice. Many refactoring techniques have been produced related to code simplicity and understandability for maintainability & extensibility. Here author enforced to make the method with the division of three sections and each section should have an argument as a signal. Such technique will be the pillar of reusability from many directions.

Keywords: Source Code, Reusability, Refactoring, maintainability and extensibility.

1. Introduction

Different opinion from different dimensions has been explored about refactoring as explained below:

"Changing the structure of a code with changing its function to make easier code"

"A change that alters the functionality of the code is not a Refactoring"

"Greatly reducing the chance of error is relate to Refactoring"

One possible definition (Sibylle Peter 2009) about code refactoring is that actually it is technique for reconstructing an existing code, without changes in external behavior and altering the internal structure. It is useful for easier to fix bugs and easy to read the source doe. Every bug leads too many new bugs and it will be very hard to implement new features, here master plan is suggested which explores what should be refactored. Also definition published in Xp Refactoring Faq, refactoring should not be fixed with only bugs fixing but it is very close to reusability, at start technical staff may not understand what is to reuse and how to reuse.

Alexandre Correa (2007) & Raimund Moser (2006) have argued that refactoring is not only supposed to improve source code understandability & maintainability but has positive effects on reusability's internal measures of object oriented programming. It also promotes ad-hoc reuse in development environment. Different techniques of refactoring can be applied over OCL (object constraint language) specification to remove problematic constructions by using UML 2.0. Refactoring opportunities for extract method Nikolaos Tsantalis (2011), where it is defined the different rules for program behavior and extraction of meaningful refactoring opportunities. This leads to either complete computation of a variable or the statements affecting the state of an object.

Bill Opdyke (1999) suggest a strategy about replacement of automated system over existing system which is not just for limited time, but automated systems replace previous one with life time. Software for an organization completely fulfills the currently running procedures, while organization requires new services time to time. Due to different approaches related to software models and programming tools & schemes, such new services can be easily accommodated. When organization requires some amendments in an existing service, then major issue for such option is Refactoring. Refactoring should be applied more generally due to different issues like platform selection; product evaluation and software reuse.

Changes in existing services are the common and burning issue from the start of organization automation. Programmers feel very difficulties by changing the source code of existing service. Article published by The-Software-Experts (2010), it is claimed that If code writers follow the principles of refactoring from the beginning of source code, then updating can be easily understandable and changeable. OOP approaches require a good architecture to integrate millions of lines of code programmed by various programming team members.

Soft Goals is very useful for refactoring, due to adding comments or regrouping functions into other modules. Since refactoring has direct link with reusability so Object Oriented Approaches (OOA) are very favorable. In OOA, class is a single entity with the combination of different methods (made up of code refactoring principles) so effects of different coupling suggested by Shakeel Ahmad (2011) about common coupling , external coupling & control coupling, they can be minimized while content coupling & data coupling can be easily handled. By using such type of emerge, refactoring make its position in programming zones.

Class is also the technique for refactoring means: extract class move part of code for an existing class into new class. Class consists of different methods and refactoring can be easily applied over each method and it can be reusable for other services or existing code can be changed. As for reusability and refactoring OOP concept had been generated, because repeated executed code should be written in single piece i.e. method. This can be easily changed for some other purpose. Method does not consist of single line of code; it is also built up of many line of code and some other methods. Here author suggest a scheme for source code to build the method or function with and without signal, where refactoring can be easily applied, if code of source-code is so long. Then a method can be embedded in other class of same tool or other tool.

2. Refactoring in Class Method with a Signal

It is common known, that each service has different activities and each activity have operational parameters and procedures. If service treats as a class then activities treat as methods. In broad vision, purpose of each activity is same. This is solid matter and universal truth for each activity. Every body knows the purpose of any activity in short words input, processing and out put. It is very clear that each function is the combination of three sections. By such division changes can be easily made.

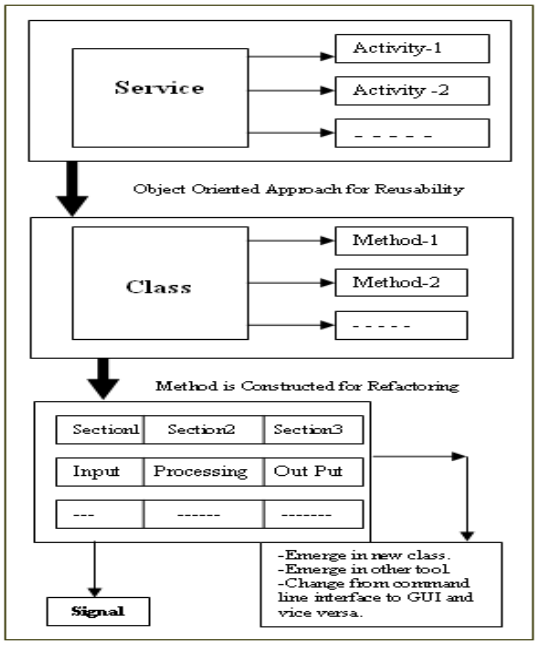


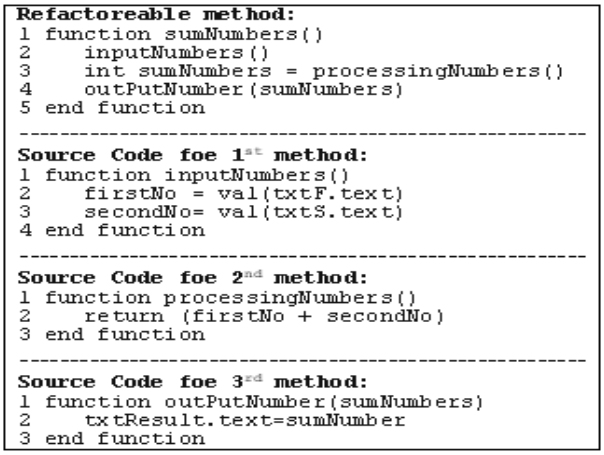
Fig-1: Method of a Class with three Sections and Signal for Refactoring

Water pump service is the good example for mapping the above diagram. Such service consists of electric machine consider as an input, water tank consider as a processing and tap consider as an out put. At any time, service consumer can change any item.

User can set any item separately, when there is a fault. If user wants to replace old thing with new one, he can change any thing with out disturbing other one i.e. if user wants to change the tap, he can easily replace it without disturbing water tanks or machine. Applying this logic on any method of a class, interface for each option can be easily fit in frame of user demand.

Fig-2: Change of any Item in Water Pump Service

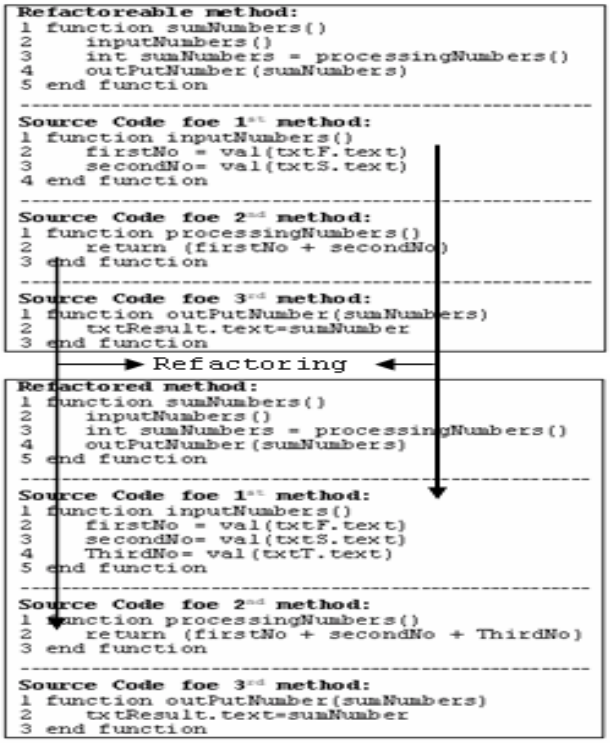
Suppose a method, which is used to add two numbers then it should be divided in to three sections.



```
Refactoreable method:
1 function sumNumbers()
2    inputNumbers()
3    int sumNumbers = processingNumbers()
4    outPutNumber(sumNumbers)
5 end function
-----------------------------------------------------
Source Code foe 1st method:
1 function inputNumbers()
2    firstNo = val(txtF.text)
3    secondNo= val(txtS.text)
4 end function
-----------------------------------------------------
Source Code foe 2nd method:
1 function processingNumbers()
2    return (firstNo + secondNo)
3 end function
-----------------------------------------------------
Source Code foe 3rd method:
1 function outPutNumber(sumNumbers)
2    txtResult.text=sumNumber
3 end function
```

Fig-3: Method of a class consists of three Sections

In above diagram, changes can be done separately by disturbing other one. Such changes may type changing, space setting for variables or some functionality not behavior. Suppose if requirement is changed from addition of two numbers to three numbers. Then only changes will be done in input section and procession section as shown in following figure.



```
Refactoreable method:
1 function sumNumbers()
2    inputNumbers()
3    int sumNumbers = processingNumbers()
4    outPutNumber(sumNumbers)
5 end function
-----------------------------------------------------
Source Code foe 1st method:
1 function inputNumbers()
2    firstNo = val(txtF.text)
3    secondNo= val(txtS.text)
4 end function
-----------------------------------------------------
Source Code foe 2nd method:
1 function processingNumbers()
2    return (firstNo + secondNo)
3 end function
-----------------------------------------------------
Source Code foe 3rd method:
1 function outPutNumber(sumNumbers)
2    txtResult.text=sumNumber
3 end function
```

Refactoring

```
Refactored method:
1 function sumNumbers()
2    inputNumbers()
3    int sumNumbers = processingNumbers()
4    outPutNumber(sumNumbers)
5 end function
-----------------------------------------------------
Source Code foe 1st method:
1 function inputNumbers()
2    firstNo = val(txtF.text)
3    secondNo= val(txtS.text)
4    ThirdNo= val(txtT.text)
5 end function
-----------------------------------------------------
Source Code foe 2nd method:
1 function processingNumbers()
2    return (firstNo + secondNo + ThirdNo)
3 end function
-----------------------------------------------------
Source Code foe 3rd method:
1 function outPutNumber(sumNumbers)
2    txtResult.text=sumNumber
3 end function
```

Fig-4: Method easily changed According to User Needs

According user requirements, programmer will easily locate the location where changes will be done. Fig-4 shows that only source code of 1st and 2nd method is changed from previous requirements to new requirements. In 1st method only line-4 is added for input of third number and in 2nd method only line-2 is changed from addition of two numbers to three numbers.

As different language provide different graphical user interface or command line interface. Adaptation of such technique, interface can be easily interchanged. User interface depend on input and out put, not on

processing. Now here separate identification has assigned to input and out put sections, so changes over here can be easily done.

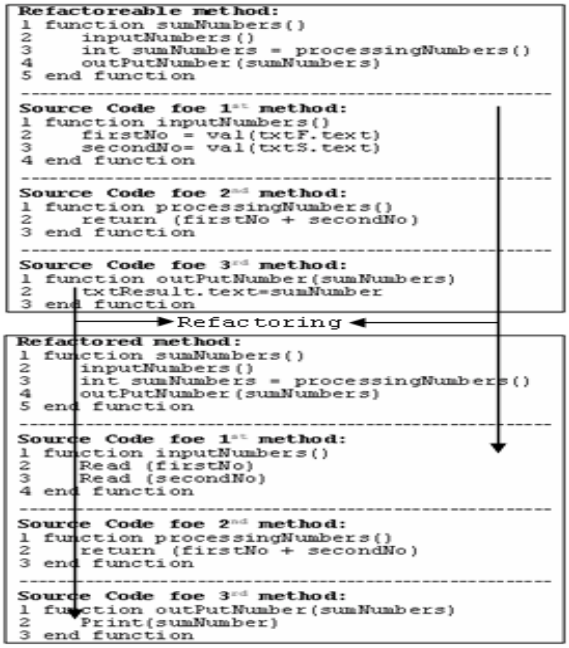


Fig-5: Graphical Interface easily changed from GUI to Command Line

From above figure it is clear that only changing the Graphical User Interface GUI-statements from 1st and 3rd methods to command line statements. There is no need to change the processing statements.

Since we have changed input or out put interfaces but in same place some users require graphical interfaces and some require command line interface. Similarly some require addition of two numbers and some requires of addition of three numbers. Then make the habit to pass a signal to each method as a parameter then requirements of everyone can be completed.

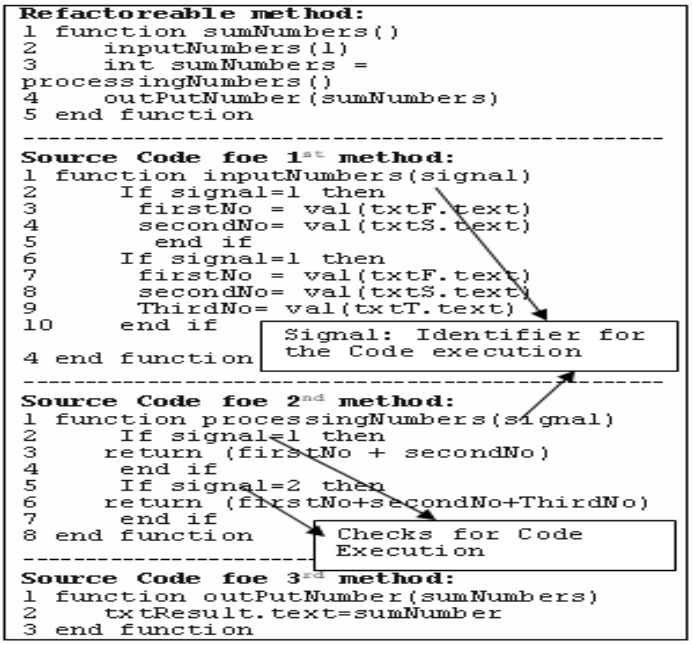


Fig-6: Method with Signal for Code Execution

By above setting of a method with signal, any type of code can be appended with function. Any user who want to add to two numbers, he will pass signal as 1 and for addition of three numbers, signal as 2 will be passed.

Some users use to add two numbers only with the desired inputs, then this class method will be easily reused with the little bit refactoring. Suppose some users wants to add two numbers which are even then only input section of a method can include a new method for checking the input constraints.

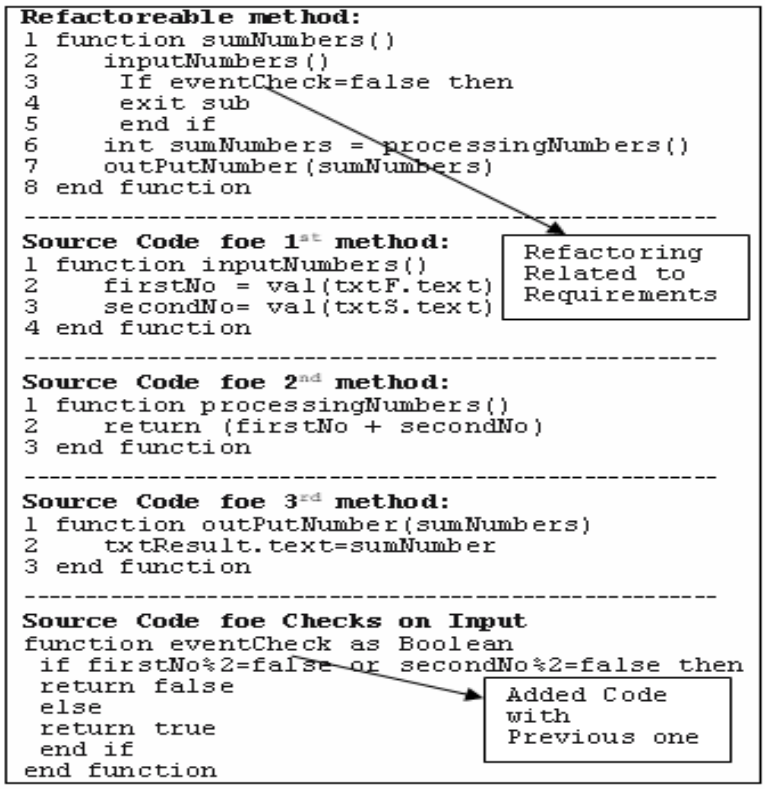


Fig-7: Adding New Requirements with Previous One

It is clear that by adding the some portion of a new method with previous one, previous method will be useable for new queries.

It is true that, we have taken simple example, but we are seeing the refactoring of a method from an angle of reusability by optimizing the code. Where we have suggested that division of each method for an activity should be in three sections and strongly recommending for passing signal through each section.

3. Conclusion

If a method has long body, then it may have a duplicate code with nearby method. Such problem can be recognized by refactoring i.e. transforming the routine in to new structure that behaves the same as before. Besides the maintainability (easy to fix the bugs) and extensibility (easier to expand the capabilities of the function), refactoring should be useful for reusability. Here author suggest a refactoring scheme for altering the code of a method of a class. By adopting such technique, a method with signal can be refactored easily for current facilities as well as future reusability. Here authors take a simple example, but purpose of proposed research is not relate to such example, but relates to technique adopting in research. Proposed research enforces each programmer to divide each method in to three sections with a passing argument as a signal. Then on any time such code can be easily transferred from graphical user interface to command line interface. And also single method can fulfill the requirements of many users.

References

Bill Opdyke , (1999), Refactoring, Reuse & Reality, *Lucent Technologies/ Bell Labs*. http://st-www.cs.illinois.edu/users/opdyke/wfo.990201.refac.html

Shakeel Ahmad, (2011), "Reusable Code for CSOA-Services: Handling Data Coupling and Content Coupling", (IJCSIS) *International Journal of Computer Science and Information Security*, Vol. 9, No. 5, 196-199.

Alexandre Correa, (2007) "Refactoring object constraint language specifications", *Software System Model*, Springer, Vol 6:113–138

Principles of Code Refactoring, (2010), *The-Software-Experts*, http://www.the-software-experts.de/e_dta-sw-refact-principles.htm

Sibylle Peter, 2009,"Refactoring Large Software Systems", *ISSN1661-402X Methods & Tools* , Vol 17, no 4

Raimund Moser, (2006), "Does Refactoring Improve Reusability?", *Springer-Verlag Berlin Heidelberg.,ICSR*, LNCS 4039, pp. 287 – 297

Nikolaos Tsantalis, (2011) "Identification of extract method refactoring opportunities for the decomposition of methods", *Journal of Systems and Software*, ELSEVIER**,** Volume 84, Issue 10, Pages 1757-1782