ISSN XXXX XXXX © 2017 IJESC

Research Article Volume 7 Issue No.5

Code Refactoring using Design Flaw Metrics

Aakash Poliyath[1], Swaraj Patel[2], Vedant Salvi[3], Suchita Patil[4]
BE Student[1, 2, 3], Assistant Professor[4]
Department of Computer Engineering
K. J. Somaiya College of Engineering, Mumbai, Maharashtra, India

Abstract:
Refactoring is a software engineering discipline that has emerged over recent years to become an important aspect of maintaining software. Refactoring is the process to restructuring of software according to specific guidelines and principles. Software system quality typically falls down as the system is subjected to new changes during the course of its lifetime. Successfully maintaining such a system demands that in addition to adding new functionality, existing code must be continuously refactored, i.e., improved without adding functionality. Existing IDE's depend more on programmer's knowledge to identify design flaws and apply refactoring changes. In this paper we have proposed an automated approach that is based on analyzing the code quality using design flaw metrics and then suggesting corresponding refactoring techniques. This automatic refactoring allows an exploratory approach to design.

Keywords: Code refactoring; design flaw metrics; software engineering; refactoring tool, code quality.

I. INTRODUCTION

The process of refactoring using conventional IDEs is as follows – Selecting the part of code that is to be refactored and then choosing a refactoring technique from the available list to be applied. The drawback of this process is that the programmer has to manually choose the code to be refactored. Also, the existing tool set for refactoring are too expensive which forces programmers to refactor programs manually. As the size of project grows, the developer would find it difficult to keep track of each and every part of the project and thus the code quality decreases. In the approach followed here we have considered various design flaws [1, 6] and their corresponding refactoring techniques. Design flaw metrics are measures used to improve the software development process and quality. For detecting design flaws in object oriented programming, we need an appropriate representation of a program into a program model. We have developed a software application that automatically parses the code into a program model, computes various design flaw metrics and then suggests refactoring techniques for the corresponding design flaw.

II. DESIGN FLAWS

We have considered following design flaws in our system:

1. Low Cohesion

Cohesion or module strength refers to the notion of a module level "togetherness" viewed at the system abstraction level [6]. Low cohesion refers to modules that have divergent and unassociated tasks. The purpose of refactoring methods or classes to be more cohesive is that it makes the code design manageable for others to use. High cohesion can cause reduced module complexity, increased system maintainability, and increased reusability of the system. For measuring cohesion, we are using Lack of Cohesion metric 5 (LCOM5) [1, 5]

LCOM5 = (a-kl)/(l-kl)

Where 'ℓ' is the number of attributes, 'k' is the number of methods, and 'a' is the summation of the number of distinct attributes accessed by each method in a class.

The value of LCOM5 ranges from 0 to 1 where cohesion decreases from 0 to 1 [4]. Cohesion can be increased by applying move method and extract class refactoring techniques [7].

2. Data class

A data class is a class that holds only fields and basic methods for accessing them (getters and setters) [6]. They act as simple containers for data to be used by other classes. They do not contain any supplementary functionality. In a data class some standard functionality is often mechanically derivable from the data. A data class cannot independently operate on the data that they have. A Data Class has a high ratio of the number of getter-setter methods in the class to the total number of methods in the class as compared to all other classes in the program, denoted as [1]

(#SM+#GM)/#M

Where #SM is number of setter methods in the class, #GM is number of getter methods in the class and #M is the total number of methods in the class. If a class has fields declared as public, encapsulate the fields to hide them from direct access and restrict access using the getter and setter methods only.

3. Large class

A Large Class is a class having relatively high number of members including fields and methods as compared to the average of all classes in the program. Large class is denoted by (#F +#M) [1], where 'F' is number of fields and 'M' is number of methods. It often contains too many instance variables that may not be required in the program.

Refactoring of Large Classes help in preventing code duplication and also improves the functionality of the code. Extract Class can be used where a part of the large class can be extracted into a separate class [7]. Extract Interface can also be used for isolating only common interfaces that are used by the client [7].

4. Controller class

A Controller class has relatively small ratio of the number of incoming methods to the number of outgoing methods as compared to the average ratio of all classes in the program [1,6]. Controller class is denoted as (#II/#OI), where 'II' is the number of incoming invocations and 'OI' is the number of outgoing invocations and this ratio is low as compared to the average ratio of all other classes in that program. This design flaw is characterized by a class diagram that consists of a single large complex controller class surrounded by simple data classes. Move Method and Move Field can be used to reduce the complexity of the code thereby creating a balance between the number of incoming and outgoing method invocations [7].

5. High Coupling

The coupling highlights the dependency between the modules or classes. Highly coupled program units are depending on each other, loosely coupled units are independent or almost independent [5]. Lower coupling is better for high quality software as modifications can be introduced in the desired class without affecting the other classes in the code. There are various types of coupling, however we have considered only the coupling between object classes denoted as [8]

CBO = (Number of links)/(Number of Classes)

Extract Class creates a new class which contains a selected group of methods and attributes from the original class [7]. This can isolate the parts of the code that depend on some other code reducing the coupling between the classes [7].

III. SYSTEM BLOCK DIAGRAM

The system contains 5 components as:

- Data extraction component- This component is used for extracting information such as number of methods, fields from the java code.
- Metric evaluation component- This module uses the data extracted from the code to calculate various flaw metrics such as LCOM5.
- Flaw detection component- Based on the metrics, flaws in the code are identified. For example, code with high LCOM5 close to 1 has low cohesion.
- Refactoring component- For the flaws present, appropriate refactoring techniques are applied. For low cohesion, Extract class refactoring technique is applied.
- Code analysis component- The refactored code is analysed to check if any errors are present or not.

This information such as number of methods, fields from the java code may be used in calculation of one or more design flaw metric. Hence it is essential to extract all these parameters first hand and then proceed to calculate design flaw metric and not extracting parameter one by one for each flaw metric. This will save computation cost and reduce the time required for computation. Also after the code is refactored it is essential to check for any possible errors in the code which is done by the code analysis component.

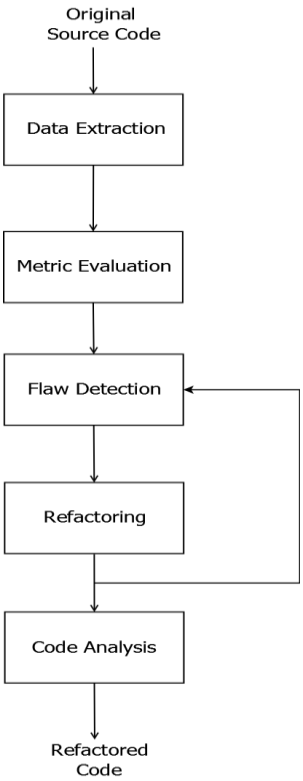

Figure.1. Block Diagram of System

IV. METHODOLOGY

To identify design flaws we need to compute various design flaw metrics for the given source code [2, 4]. Computing design flaw metrics involves entities like class, methods, fields and accesses made by an entity [5]. We need to extract information about these entities from program under consideration. This information is represented in appropriate form called as program model to be further used for design flaw computation [3]. Using program model we can hold the data corresponding to the given code and use it later to compute various design flaws and apply refactoring changes to the code [3]. The basic node that we have used to store the data for the respective class is given below:

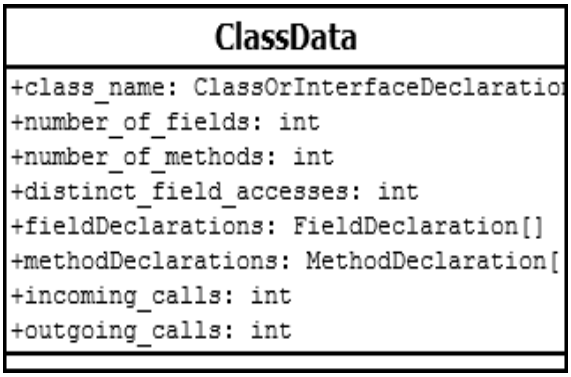

Figure. 2. Basic Node in Program Model

The object of class 'ClassData' is used to represent the data about the program entities at the class level. The overall methodology can be summarized as:

Algorithm:

1. Load the source code to be refactored.
2. Parse the code to find out:

a. Total number of Classes and their names.
b. Number of methods and fields in each class.
c. Number of instance variable accesses made by each method in class.
d. Number of getter-setter methods in the class.
e. Number of incoming and outgoing method invocation for each class.

3. Compute following design flaw Metrics:
a. LCOM5.
b. Call Ratio.
c. Data class ratio.
d. CBO.
e. Average member ratio.

4. Compare the calculated design flaw metrics with the threshold and show the corresponding design flaw.
5. For every design flaw suggest the appropriate code refactoring technique.
6. If the user approves the refactoring technique

Then refactor the code and apply changes to the source code file.
Else
Don't apply the refactoring changes to the code.

V. SOFTWARE ARCHITECTURE

The software architectural style is component based architecture. The front-end GUI was built using JavaFX. JavaFX is a software platform for creating user interfaces and building desktop applications that can run across a wide variety of devices. Software Back-end which comprises of components involving metric computation, flaw identification and methods for refactoring techniques were implemented in Java.

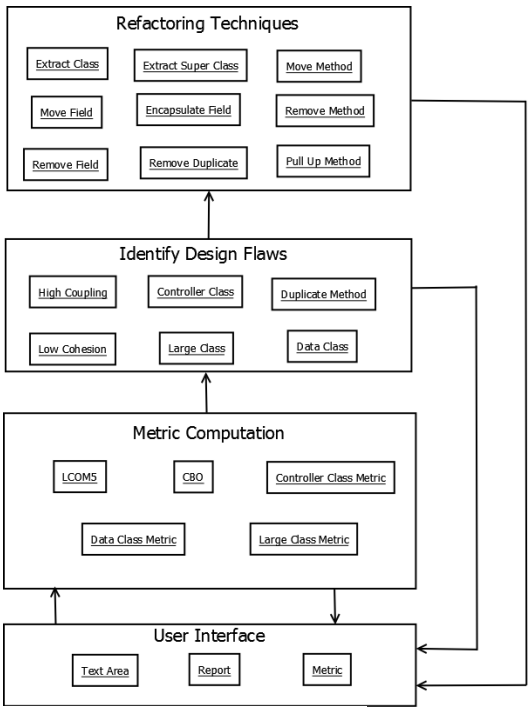

Figure.3. Software Architecture

VI. RESULTS

We went through a series of test cases and validated the refactored code with the help of design flaw metrics before and after the code refactoring was performed. The sample metrics shown below belong to test case containing three classes Staff1, Staff2 and Email. Here Staff2 has low cohesion hence its LCOM5 should be close to 1. Following image shows the metrics before the code was refactored.

Metrics			
Classes	Staff1	Staff2	Email
Number of Members	8	7	7
LCOM5	0.4	0.888888888...	0.2
Call Ratio	0.25	1.0	4.0
Data Class Ratio	0	0	0

Figure.4. Flaw Metric before Refactoring

After the code was refactored using extract class refactoring technique it was expected that LCOM5 would decrease indicating an increase in cohesion which is evident from LCOM5 metric value which dropped from 0.888 to 0.

Metrics			
Classes	Staff1	Staff2	Email
Number of Members	8	3	7
LCOM5	0.4	-0.0	0.2
Call Ratio	0.25	--	4.0
Data Class Ratio	0	1	0

Figure.5. Flaw Metric after Refactoring

VII. CONCLUSION

Design flaws in object-oriented programs (Java) affect the code quality thus increasing the risk for introducing subtle errors during software maintenance and evolution. The approach followed here presents a comprehensive technique for detecting design flaws in object-oriented programs like java. The system uses a rule-based approach for detecting different kinds of design-flaw symptoms and applies appropriate refactoring techniques.

VIII. FUTURE WORK

Automatic refactoring allows an exploratory approach to design. We have considered various design flaws and their corresponding refactoring techniques but still there is a long list of design flaws that include Code duplication, Long method, Long parameter list, different types of coupling such as content coupling, external coupling, data coupling, stamp coupling, temporal coupling etc. Refactoring techniques that can be applied are Extract Interface, Remove Parameter, and Replace parameter with method and so on. We believe that automatic refactoring tools are the best way to reduce code complexity that arises as software project evolves.

IX. REFERENCES

[1]. Sven Peldszus,Geza Kulcsar, Malte Lochau, Sandro Schulze, "Continuous Detection of Design Flaws in Evolving Object-Oriented Programs using Incremental Multi-pattern Matching", ACM, 2016, pp. 578-589.

[2]. A.Shahjahan, W.Butt and A.Ahmed, "Impact of Refactoring on Code Quality by using Graph Theory: An Empirical Evaluation", IEEE, 2015, pp. 595-600.

[3]. I.Verebi, “A Model-Based Approach to Software Refactoring”, IEEE, 2015,pp. 606-609.

[4]. Mohammad Alshayeb, "Refactoring Effect on Cohesion Metrics", IEEE, 2009, pp.3-7.

[5]. N. Rajkumar, C. Viji and S. Duraisamy, "MEASURING COHESION AND COUPLING IN OBJECT ORIENTED SYSTEM USING JAVA REFLECTION", ARPN Journal of Engineering and Applied Sciences, VOL. 10, NO. 7, APRIL 2015, pp. 3096-3101.

[6]. Martin Fowler, Kent Beck, John Brant, William Opdyke and Don Roberts, “Refactoring: Improving the Design of Existing Code,” Addison Wesley, 1999

[7]. B. Du Bois, S. Demeyer and J. Verelst, "Refactoring - improving coupling and cohesion of existing code," IEEE, 2004, pp. 144-151.

[8]. “Coupling Between Object Classes (CBO)”, http:// support. objecteering.com/objecteering6.1/help/us/metrics/metrics_in_detail/coupling_between_object_classes.htm,2017.