I.J. Information Technology and Computer Science, **2017, 6, 34-42**
Published Online June 2017 in MECS (http://www.mecs-press.org/)
DOI: 10.5815/ijitcs.2017.06.05

Recommendation of Move Method Refactoring to Optimize Modularization Using Conceptual Similarity

Md. Masudur Rahman
Institute of Information Technology, University of Dhaka, Dhaka, 1000, Bangladesh
E-mail: bit0413@iit.du.ac.bd

Md. Rayhanur Rahman and B M Mainul Hossain
Institute of Information Technology, University of Dhaka, Dhaka, 1000, Bangladesh
E-mail: rayhan@du.ac.bd, raju@du.ac.bd

***Abstract*—**Placement of methods within classes is one of the most important design activities for any object oriented application to optimize software modularization. To enhance interactions among modularized components, recommendation of move method refactorings plays a significant role through grouping similar behaviors of methods. It is also used as a refactoring technique of feature envy code smell by placing methods into correct classes from incorrect ones. Due to this code smell and inefficient modularization, an application will be tightly coupled and loosely cohesive which reflect poor design. Hence development and maintenance effort, time and cost will be increased. Existing techniques deals with only non-static methods for refactoring the code smell and so are not generalized for all types of methods (static and non-static). This paper proposes an approach which recommends 'move method' refactoring to remove the code smell as well as enrich modularization. The approach is based on conceptual similarity (which can be referred as similar behavior of methods) between a source method and methods of target classes of an application. The conceptual similarity relies on both static and non-static entities (method calls and used attributes) which differ the paper from others. In addition, it compares the similarity of used entities by the source method with used entities by methods in probable target classes. The results of a preliminary empirical evaluation indicate that the proposed approach provides better results with average precision of 65% and recall of 63% after running it on five well-known open projects than JDeodorant tool (a popular eclipse plugin for refactorings).

***Index Terms*—**Code Smell, Refactoring, Feature Envy, Move Method, Coupling, Cohesion, Conceptual Similarity.

I. Introduction

Code smell is a design problem that makes source code duplicate, tightly coupled and complex. Therefore, code smells should be removed from the application in order to make maintenance task easier. Refactoring is the technique which is used to remove the code smells by restructuring existing code [1]. As a result, it improves the software quality in terms of maintainability and reengineering process [2]. In the last decade, code smells have become an established concept for patterns or aspects of software design that may cause problems for further development and maintenance of the system [3].

Among the 22 types of code smells described by Martin Fowler [1], Feature Envy is one of those smells that is directly related to coupling and cohesion in an object oriented application. The code smell exists in the application when a method makes too many calls to other classes to obtain data or functionality (i.e., feature), in order to accomplish its task, rather than that of its current class. Moreover, this type of code smell occurs when developers violate the principle of grouping behavior with related data. This violation makes the application tightly coupled and loosely cohesive, and eventually imperfect modularization among the components. In the case of structured design and programming, application design with low coupling and high cohesion lead to products that are both more reliable and maintainable [4].

In an object oriented system, classes encapsulate internal states manipulated by their methods. However, developers often unconsciously implement methods into incorrect classes and thus create feature envy code smell that makes the application complex in terms of coupling and cohesion [1]. High levels of coupling and lack of cohesion make an application so complicated that it becomes very difficult for developers to maintain the application in the long run. In addition, during the development and maintenance phase, changing in one class makes effect in other classes that leads extra activities to change those affected classes due to high coupling and low cohesion. Coupling is a significant factor to measure complexity of the application and to analyze change impact [22]. Therefore, to maintain high quality software, developers' should implement loosely coupled and highly cohesive design [5]. Moreover, modifying existing classes as well as introducing new features require higher effort if feature envy code smell presents in the application rather than other code smells [3]. So refactoring of the code smell by moving methods

into appropriate classes from incorrect ones plays a significant role to reduce coupling and increase cohesion, and eventually, enrich modularization of the application.

However, manual inspection to group similar behaviors of methods in the same classes is a lengthy and risky process, since assumption of placement of methods might not be correct always and it varies from developer to developers. Therefore, design and maintenance problem of the application might exist in the manual process. In order to decrease coupling and increase cohesion in the application, automatic move method refactoring technique is indispensable which eventually optimizes the interactions among the modularized components. The technique is used to detect methods implemented in incorrect classes and recommend more appropriate classes for those methods. In literature, most of the techniques were based on coupling and cohesion to group similar methods in a class. The traditional approach is to recommend a method to a class whose entities are used mostly by the method rather than similar behavior. However, these existing techniques do not focus on conceptual similarity whereas a class should stands for SRP [1] (Single Responsibility Principle). The methods within a class perform the responsibility which is referred as a concept of being grouped together. This conceptual behavior is an important factor to group similar methods into a class, regardless of the directly used entities (method calls and used attributes) by methods of classes. In software design and modularization, similar behaviors of methods that perform similar tasks should be grouped together into classes to achieve optimized interactions among the modules. In other words, conceptual similarity is defined by similar entities used by the source method and the methods in a classes. Moreover, we consider both static and non-static entities (methods and attributes) in the technique whereas most of the existing works considered only non-static entities. This move method refactoring approach assists developers significantly by reducing development and maintenance effort, time and cost through improving software modules.

This paper proposes an approach of recommending move method refactoring technique based on similar concept or behavior of methods in a class. The approach consists of three phases. In the first phase, it analyzes source code information by parsing source class files and generates conceptual set. The set contains references (class names) of both static and non-static entities (method calls and attributes) used by methods. The conceptual behavior and inclusion of both static and non-static entities help to group similar methods more accurately and makes the approach different from existing ones. In the second phase, similarity between a source method and probable target classes are calculated using conceptual set. Here, Jaccard Similarity [18] Coefficient is used to calculate similarity by considering both static and non-static entities. In the third phase, by comparing the similarity values of the method's current class and other classes, it is decided whether feature envy code smell exists in the system or not. If the similarity value of the method's current class is less than the values of other one or more classes, then the approach detects the method as a feature envy code smell and suggests more appropriate class to move on. Thus, this approach refactors the code smell which also reduces coupling and increases cohesion, and eventually enriches modularization of the application.

For validation, we experiment our approach on five well-known open source java projects and compare the results with JDeodorant tool[2] (a popular eclipse plugin for refactorings). The preliminary empirical evaluation provides satisfactory results with average precision of 65% and recall of 63% which are better than JDeodorant tool. The results also indicate that the incorporation of conceptual strategy and inclusion of static entities along with non-static are important factors for recommending of move method refactoring technique to enrich software modularization.

In summary, the paper makes the following major contributions:

1) A recommendation approach of move method refactorings based on conceptual similarity to optimize software modularization.
2) A technique to automatically detect feature envy code smell for both static and non-static methods.
3) Evaluations on five popular open source java projects. The results show that our approach more effectively recommends move method refactoring than JDeodorant tool.

The remainder paper is organized as follows: Section II discusses the existing works related to feature envy code smell detection and its refactoring technique. Section III describes the proposed recommendation approach, while Section IV discusses results from a preliminary empirical evaluation. Section V shows a case study of the whole approach and finally, Section VI concludes the paper.

II. Related Work

A number of works exists in the literature regarding the identification of feature envy code smells and move method refactoring opportunities, mainly related to methods implemented in incorrect classes. These techniques are mostly based on structural information analysis from source code and historical information analysis from versioning system. These existing approaches are described in this section.

JDeodorant is a well-known eclipse plugin for refactorings that identifies five kinds of code smells, namely "Feature Envy", "Type Checking", "Long Method", "Duplicate Code" and "God class" [6], [7]. Feature envy is one of those smells that the tool identifies as well as provides recommendation to the appropriate

[1] In SRP (Single Responsibility Principle) a class should have only a single responsibility: http://www.oodesign.com/single-responsibility-principle.html. [Last Accessed 15 June, 2016]

[2] https://marketplace.eclipse.org/content/jdeodorant. [Last Accessed 12 July, 2016]

classes of the affected methods which is proposed by Fokaefs *et al.* [6]. The identification of feature envy code smell is based on the notion of distance between methods and system classes. The tool, JDeodorant follows a classical heuristic in order to detect the code smell: A feature envy code smell is identified if the distance of a method from a system class is less than the distance of this method from the class that it belongs to. The distance which can also be referred as dissimilarity between methods and system classes has been measured by *Jaccard Distance* technique. The tool also suggests more appropriate classes for the affected methods based on the distance score.

JMove is another tool for eclipse plugin which is used to refactor the feature envy code smell using 'move method' technique [8]. The approach is based on the dependency set which is calculated using coupling and cohesion. The calculation of dependency set consists of the references of attributes, parameters, return types and method calls established by a given method located in a class. Then *Sokal and Sneath 2* similarity coefficient is used to detect the smell. This technique claims better result in terms of recall than JDeodorant. However, the both techniques detect only non- static methods as feature envy code smell.

To detect feature envy code smell and use move method refactoring technique, an approach called *Methodbook* has been proposed by Oliveto *et al.* [9]. This approach uses *Facebook* as metaphor to detect the smell. It identifies the friend methods of the affected method to calculate similarity and provide recommendation to the more appropriate class based on the calculation. However, the *Methodbook* technique performs better in terms of precision whereas the number of detection is 40%. In addition, it is difficult for the *Methodbook* process to identify envied class when a method has significant similarities with almost same number of methods of multiple classes. In that case, the technique may give inefficient result.

HIST (Historical Information for Smell Detection) is an approach proposed by Palomba *et al.* in 2013 to detect five different code smells in which feature envy is one of those smells. It exploits change history information mined from versioning systems [10]. The feature envy code smell can be detected solely relying on structural information and several approaches based on static source code analysis have been proposed to detect the smell. Thus, HIST is able to compare directly to these code analysis based approaches for detecting feature envy smells to assess to what extent change history data might be of some value in the detection also of these types of smells. Considering another view is that a feature envy may manifest itself when a method of a class tends to change more frequently with methods of other classes rather than with those of the same class. Based on such consideration, HIST approach has been update in 2015 to detect smells based on change history information mined from versioning systems and specifically, by analyzing cochanges occurring between source code artifacts [11].

inCode is an eclipse plugin which is used to identify feature envy smells of static methods only [12]. It does not manipulate any data of the source class but it processes data of other system classes. According to object oriented design heuristics and principles, method must be placed in the class, in which data it manipulates more. This basic heuristic has been used in inCode approach to detect these methods as feature envy code smell [13]. Due to no access to inCodes documentation, the approach is not understandable of how it detect only static methods as smell rather than non-static methods.

Tsantalis *et al.* have proposed an approach to identify move method refactoring opportunities based on coupling and cohesion using Jaccard distance [14]. They have suggested the refactoring opportunity on the basis of certain preconditions. However, they have not considered whether the target method and the suggested class are contextual similar or not. Designer has to take the final decision by manual inspection of design documents. So, manual efforts of checking conceptual similarity are needed.

Fontana *et al.* have proposed machine learning techniques to detect several code smells including feature envy, but not suggest any refactoring opportunities [20]. Kimura *et al.* have proposed a technique to detect the refactoring candidates by analyzing method traces that contains method invocation in program execution [15]. It detects irregular methods as candidates of move method based on pattern of method invocations. Without having the method traces that is program execution, the technique will not work. In another paper, Napoli *et al.* have provided move method refactoring opportunity based on CBO (Coupling Between Objects) and LCOM (Lack of Cohesion on Methods) aiming at optimizing modularity for large systems as well as emphasized on GPU (Graphics Processing Unit) rather than single CPU (Central Processing Unit) for faster calculation [16].

As stated above, these researches has addressed the importance of removing feature envy code smell, as it occurs due to the violation of two significant design principles coupling and cohesion. Several automated approaches have been proposed throughout the years to refactor the code smell. However, almost all the techniques have used coupling and cohesion in the code smell detection and move method refactoring approaches, and have not considered the SRP that represents behavioral similarity that a method and its class stand for. Moreover, most of the cases, these techniques have avoided static entities in the detection process.

III. Proposed Approach

The proposed approach is used to provide recommendation of move method refactorings to remove feature envy code smell in any object oriented application.

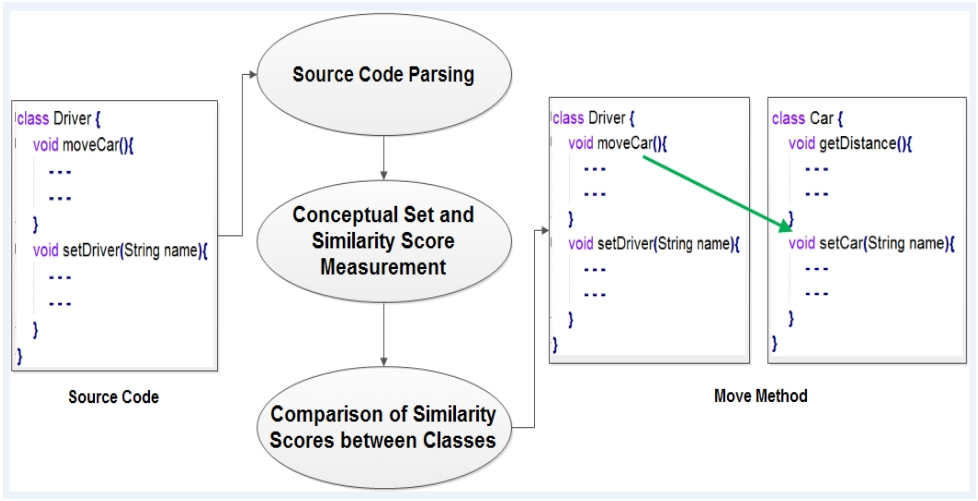

Fig.1. Architecture of Recommending Move Method Refactoring Using Conceptual Similarity

The approach identifies similar methods of a target method which should remain in the same class based on the conceptual similarity. The approach has three phases (shown in Fig. 1):

1) Source Code Parsing,
2) Conceptual Set and Similarity Score Measurement Using Jaccard Similarity Coefficient, and
3) Compare the Similarity Scores between Classes to Recommend Move Method Refactoring.

In the first phase (Source Code Parsing), in order to group similar methods into a class and recommend move method refactorings, information like - classes, methods, attributes, etc. from the source code of an application are required to be analyzed. To analyze these information, a third party parser named ByteParser [17], [21] which is used to parse java bytecode, is used in this phase. The parser analyzes information of both static and non-static method calls and used attributes by each method of classes in the application. The analyzed information from class files of the application are the basis of the approach which are used to calculate similarities between the source method and classes. So source code parsing to identify and refactor the code smell can be considered as the initial and fundamental step of the implementation part.

In the second phase (Conceptual Set and Similarity Score Measurement Using Jaccard Similarity Coefficient), from parsing phase, analyzed information - used entities (method calls and used attributes) by the source method are fed up in this phase. The main task of this phase is to calculate similarity between the source method and other methods of classes based on similar entities rather than direct used entities using Jaccard Coefficient [18]. To calculate similarity, we use class names of used entities instead of reference or object names, as object cannon be used for static method calls. Hence, the step is more accurate to group similar methods.

In the third and final phase (Compare the Similarity Scores between Classes to Recommend Move Method Refactoring), similarity scores measured in second phase are compared between the method's current class and other probable target classes of the application. The class with highest similarity score and greater than the method's current class is recommended in which the method should be moved as the refactoring technique.

In the remainder of this section, we describe the recommendation algorithm proposed in this paper (Subsection A) and the similarity calculation function that plays a central role in this algorithm (Subsection B).

A. Recommendation of Move Method Refactoring

The proposed recommendation of move method refactoring process is shown in Algorithm 1. Assume, S is a system having a set of classes. m is a target method which is implemented in a class C of the system. For each class $C_i \in S$, the algorithm determines whether m is more similar to the methods in C_i than to the methods in its original class C (line 5). Note that, the similarity function based on similar behavior of methods (i.e., SRP) deals with both static and non-static entities. If C_i satisfies the condition of the line 5, that is, C_i is more similar than C, then C_i will a probable candidate class to receive m. Such classes are inserted into a list T (line 6) as there can be multiple classes to be the candidates. Finally, the most suitable class C^r to receive m is determined by the function $bestClass(m, T)$ (line 9). The function receives the target method m and a list of candidate classes C_i. It then sorts the classes according to the similarity values of the classes and provides the most appropriate class having the highest similarity value. Thus the algorithm suggests move method refactoring in order to remove and refactor feature envy code smell from the system S.

Algorithm 1 Recommendation of Move Method Refactoring Algorithm

Input: Target system S
Output: A list of candidate classes

```
1:  for each method m ∈ S  do
2:      C ← getClass(C)
3:      T ← null
4:      for each class Ci ∈ S  do
5:          if similarity (m, Ci) > similarity (m, C) then
6:              T ← T + Ci
7:          end if
8:      end for
9:      C^r ← bestClass(m, T)
10: end for
```

similarity(*m, C*) is the key function of the algorithm that computes the similarity between method *m* and the methods in class *C*. This function is described in the following subsection.

B. Similarity Measurement Function

The function relies on the conceptual set of entities (method calls and used attributes) established by a method *m* to compute its similarity with the methods in a class *C*, as described in Algorithm 2.

Algorithm 2 Similarity Measurement algorithm

Input: Target method *m* and a class *C*
Output: Similarity coefficient between *m* and *C*

```
1:  for each method mi ∈ C  do
2:      if mi ≠ m then
3:          similarityScore ← similarityScore +
            getSimilarity(m, mi)
4:      end if
5:  end for
6:  if m ∈ C then
7:      averageSimilarityScore ←
        averageSimilarityScore / [NOM (C) − 1]
8:  else
9:      averageSimilarityScore ←
        averageSimilarityScore / [NOM (C)]
10:  end if
11:  return averageSimilarityScore
```

Initially, we compute the similarity between *m* and each method *mi* in *C* (line 3). In the end, the similarity between *m* and *C* is defined as the arithmetic mean of the similarity coefficients computed in the previous step. In this algorithm, NOM(C) denotes the number of methods in a class *C* (lines 7 and 9).

The key function is *getSimilarity*(*m,mi*) in Algorithm 2, which computes the similarity between the sets of entities established by the two methods (line 3). The similarity is measured by the use of the Jaccard similarity coefficient which is defined as:

$$getSimilarity(m, mi) = \frac{A_m \cap A_{mi}}{A_m \cup A_{mi}} \quad (1)$$

Here,

A_m = set of entities used by method m
A_{mi} = set of entities used by method m_i

IV. Implementation and Result Analysis

To assess the proposed approach, preliminary experiments have been conducted on recommendation of move method refactoring. A prototype of the proposed algorithm has been implemented in java language for this purpose. The existing refactoring tool JDeodorant which is a well-known eclipse plugin, has also been used for comparative analysis. For the justification of correctness, heuristics regarding the refactoring technique from Martin Fowler is followed [1].

A. Environmental Setup and Implementation

As mentioned earlier, the algorithm for evaluation has been implemented in java programming language. The equipments used to develop the algorithm are as follows:

- Eclipse Mars version-4.5 [19]
- ByteParser

To implement the proposed algorithm in java language, Eclipse has been used. A source code parser named ByteParser has also been included in this implementation in order to analyze the input source code. ByteParser is an analyzer which is used to analyze java source code of *.class* files. It analyzes information like – class name, method name, field name, method call, etc. from the source code.

For the validation of the approach, five open source java projects have been used as datasets. The descriptions of the projects are shown in Table 1 consisting of five columns. The columns represent the project name, project version, number of class (NOC), number of method (NOM), and line of code (LOC) respectively. Each project has a large amount of NOC, NOM and LOC. From the table, it is seen that *Weka* is the largest project and *Maven* is the smallest one on the basis of NOC, NOM an LOC.

Table 1. Experimental Projects

Project	Version	NOC	NOM	LOC
JHotDraw	7.6	674	6,533	80,536
ArgoUML	0.34	1,291	8,077	67,514
JMeter	2.5.1	940	7,990	94,778
Maven	3.0.5	647	4,888	65,685
Weka	3.6.9	1,535	17,851	272,611

B. Preliminary Results

The results of the proposed approach is shown in Table 2 consisting of five columns. The table columns are – Project name, Total Instances# (total actual number of affected methods for move method refactoring), True Positive# (TP, number of methods suggested move

method refactoring correctly), False Positive# (FP, number of incorrect suggestions), and Precision (=*TP/(TP+FP)*). Our approach gets highest precision of 79% with *JHotDraw* project and lowest precision of 58% with *Weka* project which are significant in the recommendation approach.

Table 2. Results of the Proposed Approach

Project	Total Instances#	True Positive#	False Positive#	Precision (%)
JHotDraw	19	15	4	79
ArgoUML	31	16	8	67
JMeter	10	5	4	56
Maven	16	6	3	67
Weka	29	14	10	58

C. Comparative Analysis

The comparative results between the proposed approach and the eclipse plugin JDeodorant have been shown in this section. The comparative analysis shows a significant contribution of the proposed approach over other techniques. The existing approaches considered only non-static entities as they used reference names of used entities by methods to calculate similarity. But non static entities are used directly using class names instead of references. Consideration of both static and non-static entities in our proposed approach have made a meaningful improvement in case of similarity measurement. In addition, the approach has excluded the primitive types in case of similarity measurement process as those types are not related to coupling and cohesion. Moreover, the primitive types are not associated with method placement. Moreover, the approach does not follow the traditional approach that a method should be placed in the class whom entities it used mostly. Rather the approach is based on the concept that a method should be placed in the class such that the source method along with the methods of the class use similar entities. The comparative results in terms of precision and recall (=TP/(TP+FN)) between the two approaches have been shown in Table 3.

Table 3. Comparison between Proposed Approach and Jdeodorant

	Precision (%)		Recall (%)	
Project	Proposed	JDeodorant	Proposed	JDeodorant
JHotDraw	79	26	83	51
ArgoUML	67	60	70	56
JMeter	56	15	63	60
Maven	67	23	40	46
Weka	58	7	58	65
Average	65	26	63	56

The conceptual set is a vital part of the technique in which references of method calls and used attribute are listed. Both static and non-static method calls are considered in the list while other techniques use only non-static part. Therefore, the proposed technique provides better results than JDeodorant tool. It has precision of 65% and recall of 63% on average of the five projects while JDeodorant has only 26% and 56% respectively. As the technique applies similarity technique of the affected method rather than counting traditional method calls and used attributes of other classes to detect the method's appropriate class, it reduces false positive results. The results of comparative analysis are graphically shown in Fig. 2 (comparison of precisions) and Fig. 3 (comparison of recalls).

Fig.2 shows that the proposed approach provides better accuracy in terms of precision in each case of all the source projects. In Fig. 3, the approach also provides better accuracy in terms of recall in almost all cases, except for *Maven* and *Weka* project. However, the results for both of the projects are almost similar as JDeodorant tool. The highest accuracy in terms of both precision and recall that our approach has gained is for *JHotDraw* project.

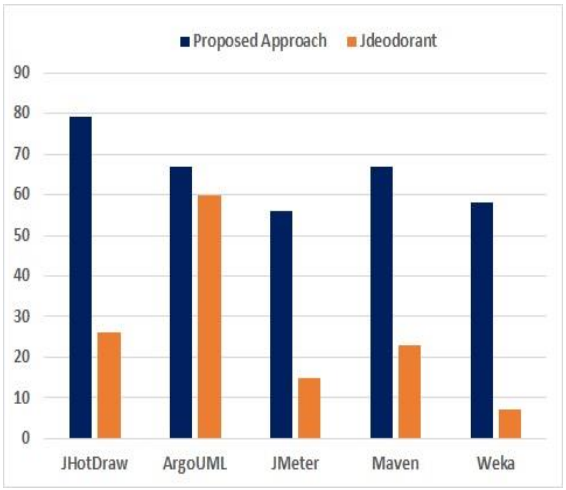

Fig.2. Comparison of Precisions

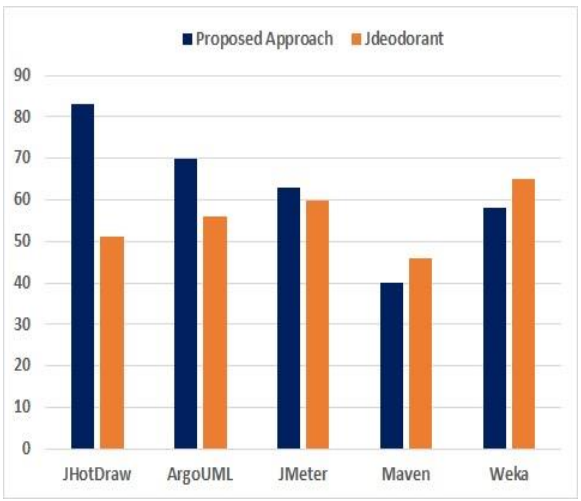

Fig.3. Comparison of Recalls

V. A Case Study on Project 'MovieRentalStore'

The result analysis of recommending move method refactorings for feature envy code smell shows the

proficiency of the proposed approach. However, a step-by-step study may increase the understanding of the approach as well as justify the reason behind the improved results than conventional approaches. Thereby, this chapter provides a phase-to-phase analysis of the recommendation approach.

For an assessment of the competency of the approach, the technique has been used on a simple java project *MovieRenalStore*. The source code has been collected from the book *Refactoring: Improving the Design of Existing Code* [3]. The main project named '*VideoStore*' in the book is free from code smell as well as refactored. So, some modification has been done in the project (*MovieRenalStore - Version2*) by injecting feature envy code smell in order to analyze the proposed approach.

The main phases of the case study are:

a. Conversion of Source Code
b. Analysis of Byte Code
c. Generation of Conceptual Set
d. Similarity Coefficient Measurement and
e. Recommendation of Move Method Refactoring

Each phase is described in the following subsections.

A. Conversion of Source Code

To recommend move method refactorings, methods and attributes in the source code are required to be analyzed. By analyzing the source code, entities like method calls and used attributes of a method are determined in order to calculate the similarity of the method with other methods in a class. So source code parsing to identify methods and attributes on the classes can be considered as initial and fundamental step of the implementation part.

A third party parser named *ByteParser* is used in order to analyze the source code. First of all, classes of the source project are converted into byte codes through compiling, that is, from *.java* files to *.class* files and made those *.class* files in *.txt* form to make the files readable or parsable using the following command –

"javap -c -private Customer.class > Customer.txt"

The sample of the source file in *.java* form and corresponding byte file in *.class* form are shown in Fig. 4 and Fig. 5 respectively.

B. Analysis of Byte Code

After conversion to the byte code classes from the base source code, those byte codes have been considered now in the form to be analyzed. Those codes have been then analyzed to get the methods and the classes of the source application for further analysis. Finally, method calls and attribute usages by a method are identified in this parsing phase of the feature envy code smell detection process.

C. Generation of Conceptual Set

After the parsing stage, conceptual set of each method of the *MovieRentalStore* project has been generated. The set consists of the references (class names) of method calls and used attributes. The conceptual set of the method *getMovie()* is: {*Movie*, *Rental*} (Fig. 6).

```
package rentalStore;
import java.util.Enumeration;
import java.util.Vector;

class Customer {
    private String _name;
    private Vector<Rental> _rentals = new Vector<Rental>();

    public Customer(String name) {
        _name = name;
    }
    public String getMovie(Movie movie) {
        Rental rental = new Rental(new Movie("", Movie.NEW_RELEASE), 10);
        Movie m = rental._movie;
        return movie.getTitle();
    }
    public void addRental(Rental arg) {
        _rentals.addElement(arg);
    }
    public String getName() {
        return _name;
    }
```

Fig.4. Source Code Example (*Customer.java* Partial)

```
Compiled from "Customer.java"
class rentalStore.Customer {
  private java.lang.String _name;

  private java.util.Vector<rentalStore.Rental> _rentals;

  public rentalStore.Customer(java.lang.String);
    Code:
       0: aload_0
       1: invokespecial #14                 // Method java/lang/Object."<init>":()V
       4: aload_0
       5: new           #17                 // class java/util/Vector
       8: dup
       9: invokespecial #19                 // Method java/util/Vector."<init>":()V
      12: putfield      #20                 // Field _rentals:Ljava/util/Vector;
      15: aload_0
      16: aload_1
      17: putfield      #22                 // Field _name:Ljava/lang/String;
      20: return
```

Fig.5. Byte Code Example (*Customer.class* Partial)

```
public String getMovie(Movie movie) {
    Rental rental = new Rental(new Movie("", Movie.NEW_RELEASE), 10);
    Movie m = rental._movie;
    return movie.getTitle();
}
```

Fig.6. Method: *getMovie*()

D. Similarity Coefficient Measurement

Similarity coefficient has been measured based on the conceptual set of method calls and used attributes using *Jaccard Similarity Coefficient* method. The measurement process is described in Fig. 7 as a flowchart.

The similarity coefficient of the method *getMovie()* in its current class *Customer* is less than another class *Rental*. So the proposed approach considers this method as a feature envy code smell.

The *getMovie()* method in the *MovieRentalStore* system is used to provide the names of movies that have been rented. So this method is not in correct class of *Customer*. So, conceptually it should be in the *Rental* class.

E. Recommendation of Move Method Refactoring

The proposed approach not only detects the method *getMovie()* as a feature envy code smell but also provides a suggestion to its appropriate class. To do this, a list has been maintained consisting of candidate classes in which the method should be moved based on the higher similarity values than its current class. The list is then sorted in descending order of the similarity values. The first class of the list having the highest similarity value is then recommended as the method's appropriate class. In this example, the *Rental* class has been suggested in which the method *getMovie()* should be moved from its current class *Customer*.

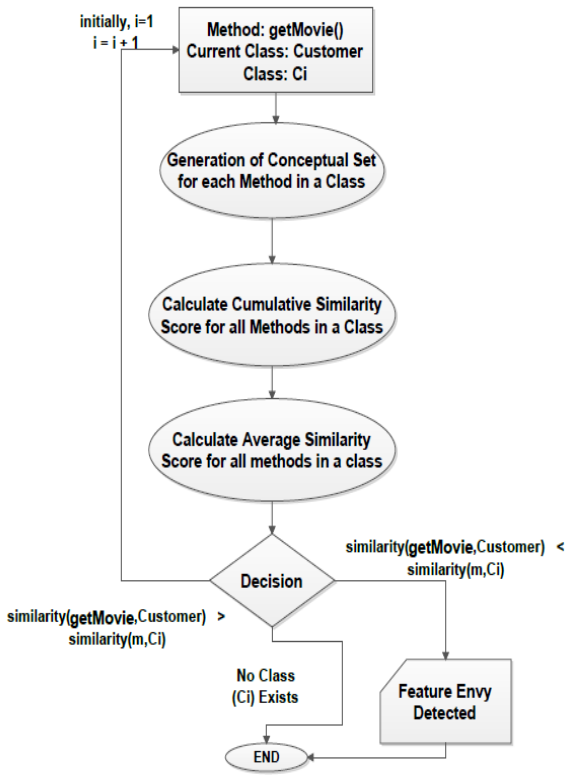

Fig.7. Flow Chart of Similarity Coefficient Measurement for Method *getMovie()*

VI. Conclusion

Coupling and cohesion are the two key factors considered during the designing phase of software application. Since developers only need to focus on coupled classes to meet a change requirements, the application should be loosely coupled and highly cohesive to make maintenance task easier with lower effort, cost and time. Feature envy code smell is a barrier to achieve this goal of maintenance task as it increases coupling and decreases cohesion. The proposed approach plays a significant role to refactor the code smell by recommending appropriate move method refactoring technique automatically. The approach based on similar behavior of methods of both static and non-static entities (method and attributes) improves the recommendation accuracy. The preliminary investigation provides satisfactory results with better precision and recall than competitive tool. In addition, the approach enriches software modularization through optimizing interactions among the components of the application. Therefore, low coupling and high cohesion are achieved.

References

[1] F. Martin, B. Kent, and B. John, "Refactoring: improving the design of existing code," *Refactoring: Improving the Design of Existing Code*, 1999.

[2] S. Demeyer, S. Ducasse, and O. Nierstrasz, *Object-oriented reengi- neering patterns*. Elsevier, 2002.

[3] D. Sjoberg, A. Yamashita, B. C. D. Anda, A. Mockus, and T. Dyba, "Quantifying the effect of code smells on maintenance effort," *Software Engineering, IEEE Transactions on*, vol. 39, no. 8, pp. 1144– 1156, 2013.

[4] N. Fenton and J. Bieman, *Software metrics: a rigorous and practical approach*. CRC Press, 2014.

[5] S. Sharma and S. Srinivasan, "A review of coupling and cohesion metrics in object oriented environment," *International Journal of Computer Science & Engineering Technology (IJCSET)*, vol. 4, no. 8, 2013.

[6] M. Fokaefs, N. Tsantalis, and A. Chatzigeorgiou, "Jdeodorant: Identi- fication and removal of feature envy bad smells," in *ICSM*, pp. 519– 520, 2007.

[7] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "Jdeodorant: Iden- tification and removal of type-checking bad smells," in *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*, pp. 329–331, IEEE, 2008.

[8] V. Sales, R. Terra, L. F. Miranda, and M. T. Valente, "Recommending move method refactorings using dependency sets.," in *WCRE*, vol. 20, p. 13, 2013.

[9] R. Oliveto, M. Gethers, G. Bavota, D. Poshyvanyk, and A. De Lucia, "Identifying method friendships to remove the feature envy bad smell (nier track)," in *Proceedings of the 33rd International Conference on Software Engineering*, pp. 820–823, ACM, 2011.

[10] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "Detecting bad smells in source code using change history information," in *Automated software engineering (ASE), 2013 IEEE/ACM 28th international conference on*, pp. 268–278, IEEE, 2013.

[11] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Mining version histories for detecting code smells," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 462– 489, 2015.

[12] R. Marinescu, G. Ganea, and I. Verebi, "incode: Continuous quality assessment and improvement," in *Software Maintenance and Reengi- neering (CSMR), 2010 14th European Conference on*, pp. 274–275, IEEE, 2010.

[13] A. Hamid, M. Ilyas, M. Hummayun, and A. Nawaz, "A compara- tive study on code smell detection tools," *International Journal of Advanced Science and Technology*, vol. 60, pp. 25–32, 2013.

[14] N. Tsantalis and A. Chatzigeorgiou, "Identification of

move method refactoring opportunities," *IEEE Transactions on Software Engineer- ing*, vol. 35, no. 3, pp. 347–367, 2009.
[15] S. Kimura, Y. Higo, H. Igaki, and S. Kusumoto, "Move code refactor- ing with dynamic analysis," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pp. 575–578, IEEE, 2012.
[16] C. Napoli, G. Pappalardo, and E. Tramontana, "Using modularity metrics to assist move method refactoring of large systems," in *Complex, Intelligent, and Software Intensive Systems (CISIS), 2013 Seventh International Conference on*, pp. 529–534, IEEE, 2013.
[17] https://github.com/rifatbit0401/ByteParser [Last Accessed: 15 March, 2016]
[18] https://en.wikipedia.org/wiki/Jaccard_index [Last Accessed: 10 February, 2016]
[19] https://www.eclipse.org/mars [Last Accessed: 10 November, 2015]
[20] F. A. Fontana, M. V. Mika, M. Zanoni, and A. Marino. "Comparing and experimenting machine learning techniques for code smell detection." *Empirical Software Engineering* 21, no. 3 (2016): 1143-1191.
[21] A. Satter, A. S. Ami, and K. Sakib. "A Static Code Search Technique to Identify Dead Fields by Analyzing Usage of Setup Fields and Field Dependency in Test Code." *CDUD 2016–The 3rd International Workshop on Concept Discovery in Unstructured Data*. 2016.
[22] B. Isong, and O. Ekabua,"A Framework for Effective Object-Oriented Software Change Impact Analysis," *International Journal of Information Technology and Computer Science (IJITCS)*, vol.7, no.4, pp.28-41, 2015.

Authors' Profiles

Md. Masudur Rahman is a graduate student at the Institute of Information Technology (IIT), University of Dhaka, Bangladesh. Currently, he is pursuing his Master of Science in Software Engineering (MSSE). He earned his Bachelor of Science in Software Engineering (BSSE) from the same institution. His core areas of interest are software engineering, code smell and refactoring.

Md. Rayhanur Rahman is a Lecturer at the Institute of Information Technology (IIT), University of Dhaka, Bangladesh. He received his Bachelor of Information Technology, major in Software Engineering (BSSE) and Master of Science in Software Engineering (MSSE) from the same institution. He has the experiences of working both in industry and academia. His core areas of interest are cloud computing, software engineering, security and testing.

Dr. B. M. Mainul Hossain is Assistant Professor at the Institute of Information Technology (IIT), University of Dhaka, Bangladesh. He received his Ph.D. degree in computer science from University of Illinois at Chicago, USA. Before that, he earned his Bachelor of Science and Master degrees from the department of Computer Science & Engineering, University of Dhaka, Bangladesh. He has the experiences of working both in industry and academia. He worked as a Software Engineer in Microsoft Corporation (Redmond, USA) & Accenture Technology Lab (Chicago & California). His core areas of interest are software engineering, security, data mining and machine learning.

How to cite this paper: Md. Masudur Rahman, Md. Rayhanur Rahman, B M Mainul Hossain,"Recommendation of Move Method Refactoring to Optimize Modularization Using Conceptual Similarity", International Journal of Information Technology and Computer Science(IJITCS), Vol.9, No.6, pp.34-42, 2017. DOI: 10.5815/ijitcs.2017.06.05