Received: 28 October 2017 | Revised: 24 March 2018 | Accepted: 24 April 2018
DOI: 10.1002/spe.2591

RESEARCH ARTICLE

MMRUC3: A recommendation approach of *move method* refactoring using coupling, cohesion, and contextual similarity to enhance software design

Md. Masudur Rahman | Rashed Rubby Riyadh | Shah Mostafa Khaled | Abdus Satter | Md. Rayhanur Rahman

Institute of Information Technology, University of Dhaka, Dhaka, Bangladesh

Correspondence
Md. Masudur Rahman, Institute of Information Technology, University of Dhaka, Dhaka, Bangladesh.
Email: bit0413@iit.du.ac.bd

Funding information
ICT Division, Government of the People′s Republic of Bangladesh, Bangladesh, Grant/Award Number: 56.00.0000.028.33.065.16-747, 21-06-2016 and 56.00.0000.028.33.065.16-747, 14-06-2016

Summary

Placement of methods is one of the most important design activities for any object-oriented application in terms of coupling and cohesion. Due to method misplacement, the application becomes tightly coupled and loosely cohesive, reflecting inefficient design. Therefore, a feature envy code smell emerges from the application, as many methods use more features of other classes than its current class. Hence, development and maintenance time, cost, and effort are increased. To refactor the code smell and enhance the design quality, *move method* refactoring plays a significant role through grouping similar behaviors of methods. This is because the manual refactoring process is infeasible due to the necessity of huge time and most of the existing techniques consider only coupling-based and/or cohesion-based information of nonstatic entities (methods and attributes) for the recommendation. However, this article proposes an approach that uses contextual information, based on information retrieval techniques, along with dependency (coupling and cohesion)-based information of the application for the recommendation. In addition, the approach incorporates both static and nonstatic entities in the recommendation process. For validation, the approach is applied on seven well-known open source projects. The results of the experimental evaluation indicate that the proposed approach provides better results with an average precision of 18.91%, a recall of 69.91%, and an F-measure of 29.77% than the JDeodorant tool (a widely used eclipse plugin for refactorings). Moreover, this article establishes several relationships between the accuracy of the approach and project standards and sizes.

KEYWORDS
cohesion, contextual similarity, coupling, feature envy code smell, move method refactoring

1 | INTRODUCTION

Code smell is a design problem that makes a software application complex and difficult to maintain.[1] The existence of the code smells does not hamper the application's performance or accuracy but makes the code hard to maintain in the long run. 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 application.[2] To make the code easy to maintain and adapt with changes, code smells should be removed from the application. Usually, developers are encouraged to apply refactoring, which is a technique used to remove the code smells by restructuring the existing code. It not only increases aspects of software quality but also improves productivity[3] by making the code easy to understand and incorporate required changes. In essence, refactoring is an important weapon for engineers to ease maintenance activities through removing the code smells.

In object-oriented (OO) programming, methods that jointly implement a particular feature should be kept in the same class.[1] It assists in improving code quality by increasing cohesion and decreasing coupling in the source code. If methods are not placed in the proper classes, the feature envy code smell occurs. Usually, this smell occurs when a method uses more features (data or functionality) of other classes to accomplish its task rather than of its current class.[1] Its existence causes a severe maintenance problem by increasing coupling and decreasing cohesion. Since high coupling and low cohesion make difficult consistent changes in the artifacts of the application,[2,4] development and maintenance effort, time, and cost will be increased. Again, faults may occur in those artifacts when developers miss areas of the codes that are needed to be consistently changed after changes are done on the methods displaying the smell.[5] In order to remove the smell, the *move method* refactoring technique is commonly used. The technique moves methods from irrelevant classes to more relevant ones without changing the behavior of the application. Applying the technique manually in a large software system is a time-consuming and an error-prone task. The reason is that it is not feasible to scrutinize a huge amount of the source code carefully to group functionally related methods. Therefore, an automatic *move method* refactoring technique is inevitable, which assists in detecting methods placed in inappropriate classes and recommending more appropriate classes for those methods to be placed.

Most of the techniques in the literature were based on coupling and cohesion to find similar methods for the recommendation.[6-9] None of the research studies have addressed contextual factors of the application in the *move method* refactoring recommendation field. However, contexts of a method provide significant information and an important factor to group the similar behavior of methods. According to the single responsibility principle (SRP),[10] a class stands for a single responsibility, and the methods within the class perform the responsibility. This responsibility is referred to as a context in this article, and the methods within a class are based on it. Therefore, the contextual information might be a significant factor to improve the accuracy of the recommendation of the *move method* refactoring approach. It is noted that entities are of two kinds, namely, static and nonstatic, which are used to measure coupling and cohesion of an application. By definition, a static entity belongs to a class, and a nonstatic entity belongs to an object of a class. Therefore, a nonstatic entity can only be used on an object of a class that it belongs to. A static method can, however, be used both on the class as well as an object of the class. Moreover, a static method can access only static entities, and a nonstatic method can access both static and nonstatic entities because at the time when the static method is called, the class might not be instantiated. In the other case, a nonstatic entity can only be used when the class has already been instantiated. Since the static entity refers to the class, no object of the class is needed to instantiate, and so, the syntax to call or refer to a static entity is *className.MethodName* or *className.AttributeName*. On the other hand, to use a nonstatic entity, an object of the class needs to be created. However, most of the existing techniques have considered only nonstatic entities (methods and attributes) of a class, as they have used reference names of method invocations in the similarity measurement process. On the other hand, these research studies have not considered the static entities, as these entities are not used by references; rather, these are used by class name directly in the OO system. Missing these important factors for grouping methods leads to less accuracy in the existing approaches. Hence, it is still an open research issue on improving the accuracy of the automatic *move method* refactoring approach.

This article proposes a new approach of recommending *move method* refactoring, based on contextual similarity along with coupling and cohesion. The metric *contextual similarity* is introduced in this article to provide more accurate recommendations than existing approaches. It establishes the key development and design concept of grouping the similar behavior of methods in the same class. It depends on the contextual factor that is the textual information in the method or class body excluding programming keywords. The reason for ignoring programming keywords is that these words do not relate with the context. The integration of the metric with coupling and cohesion improves the similarity measurement process and the recommendation approach. In a nutshell, the approach is based on three factors (*C3*): coupling, cohesion, and contextual information of the application. By analyzing the *C3* factors, it combines two types of similarity scores to get the total (or actual) similarity score between a target method (assuming the method is placed in an inappropriate class) and a class: (1) dependency (coupling and cohesion)-based similarity score using the Jaccard similarity coefficient and (2) context-based similarity score using cosine similarity, which is an information retrieval (IR) technique generally used for recommender systems[11,12] based on term frequency (*tf*) and inverse document frequency (*idf*) to identify a similar context. By comparing the scores between the method and each class, it is decided whether the feature envy code smell exists in the system or not. If the similarity score of the method's current class is less than the scores of other one or more classes, then the approach detects the method as a feature envy code smell. Finally, it recommends the class having the highest similarity score with the method for refactoring.

For validating the approach, a framework named “Move Method Refactoring Using Coupling, Cohesion, and Contextual Similarity” (MMRUC3) was developed, and a comparative analysis was conducted with the JDeodorant tool (a widely used eclipse plugin for refactorings). For the experimentation, the approach was applied on seven well-known open source projects.[13] The preliminary evaluation has shown satisfactory results with an average precision of 18.91%, a recall of 69.91%, and an F-measure of 29.77%, whereas JDeodorant gets 15.93%, 58.31%, and 25.02%, respectively. The results also indicate that the incorporation of the contextual strategy along with coupling and cohesion and the inclusion of static entities with nonstatic ones are important factors for the recommendation of the *move method* refactoring technique. Moreover, this article establishes several relationships between the accuracy of the approach and project standards (ie, how well a project is written) and sizes (ie, line of codes [LOCs], number of methods [NOMs], and number of classes [NOCs]). This article infers that there exists no or few relationships between the sizes of projects and the accuracy of the MMRUC3 approach but that, rather, there exists a strong relationship between project standards and the approach.

In summary, this article makes the following major contributions.

1. A novel recommendation approach of *move method* refactoring that incorporates context-based information with coupling and cohesion. The technique is novel in the sense that, to the best of the authors' knowledge, no other approach considers contextual information for *move method* recommendation.
2. Establishment of relationships between the accuracy of the proposed approach and two metrics: project standards and sizes.
3. Evaluations on seven well-known open source projects. The results provide evidence that the approach recommends *move method* refactorings more effectively than the JDeodorant tool.

The article is organized as follows. Section 2 discusses the existing works related to feature envy code smell detection and its refactoring techniques. Section 3 describes the proposed approach, whereas Section 4 shows a case study of the whole process of the approach. Section 5 discusses results from a preliminary empirical evaluation, and finally, Section 6 concludes the article with a future direction.

2 | RELATED WORK

Move method refactoring to remove the feature envy code smell is a significant research field that enhances code quality in terms of coupling and cohesion. Several works exist in the literature regarding the identification of feature envy code smells, *move method* refactoring opportunities, and context-based refactorings. These existing approaches are described in this section.

2.1 | *Move method* refactorings to remove feature envy code smells

Move method refactoring plays a significant role in detecting and removing the feature envy code smell. Several tools and approaches have been proposed in the literature for refactoring the smell automatically. Significant techniques for refactoring the smell are discussed below.

Fokaefs proposed an eclipse plugin named JDeodorant to identify the feature envy code smell and provide recommendation to the appropriate classes of the misplaced methods. The approach is based on the notion of distance between methods and system classes measured by the Jaccard distance coefficient. The distance between the target method and a class expresses the dissimilarity between the set of entities (method calls and used attributes) accessed by the method and the set of entities belonging to the class. Tsantalis and Chatzigeorgiou showed the qualitative analysis of the refactoring suggestions implemented in the JDeodorant tool.[14]

JMove, another refactoring tool for eclipse plugin, is used to refactor the code smell using the *move method* technique.[7] The approach is based on the dependency set calculated using coupling and cohesion. The dependency set consists of the references of attributes, parameters, return types, annotations, and method calls established by a given method belonging to a class. Next, the *Sokal and Sneath 2* similarity coefficient is used to refactor the smell.

Oliveto et al proposed an approach called Methodbook for identifying *move method* refactoring opportunities.[15] This approach uses *Facebook* as a metaphor to detect the smell. It identifies the friend methods of the target method to recommend more appropriate classes based on method calls, variables, and comments. However, it is difficult for the Methodbook approach to identify an envied class when a method has significant similarities with almost the same number of methods in multiple classes. In that case, the technique might give an inefficient result.

HIST (Historical Information for Smell deTection) is an approach proposed by Palomba et al to detect the code smell based on the changed history information.[16,17] This information is mined from versioning systems, specifically by analyzing co-changes occurring between source code artifacts.

inCode, an eclipse plugin, is used to identify feature envy smells, which does not manipulate any data of the source class, but it processes data of other system classes.[18] According to OO design heuristics and principles, the method must be placed in the class in which data are manipulated more. This basic heuristic has been used in the inCode approach to detect the code smell, but it does not detect the nonstatic methods detected by JDeodorant, as shown in the work of Hamid et al.[19] Due to the availability of inCode's documentation for its bad smell detection rules and metrics threshold, the approach lacks understandability. However, the inCode approach needs more improvements in its methodology to detect feature envy code smells.

Liu et al proposed an approach for identifying *move method* refactoring opportunities on a group of methods in a single class having the highest similarity and the strongest relationship among those methods.[8] The rationale behind this approach is that if two methods are strongly coupled and closely related in the business logic, when one of those methods is moved, the other may deserve a movement as well.

In another paper, Simon et al provided a visualization approach about *move method* refactoring using the distance-based cohesion metric of a system.[9] The approach has followed the design concept that methods having low distances within classes are cohesive, whereas methods having higher distances are less cohesive. Since the approach calculates cohesion between each two entities (attributes and methods) of the system, it might be a time-consuming calculation for large systems. In addition, it provides a visualization of the target entity along with other entities showing the geometric distances, resulting in a manual intervention to identify the refactoring opportunities.

Kimura et al proposed a technique for identifying *move method* refactoring candidates by using dynamic source code analysis rather than static analysis.[20] More specifically, the technique analyzes method traces containing method invocations during program execution. It detects irregular methods as the refactoring candidates based on patterns of method invocations at run time. The researchers are concerned with the fact that the quality of the source code increases by moving these methods to appropriate classes as they cooperate with one another in a program execution. However, without having the method traces, that is, program execution, the technique does not work.

For large software systems, computing metrics comprising thousands of classes or more can be a time-consuming task when performed on a single CPU. For this reason, Napoli et al proposed a solution that computes the metrics by resorting to the GPU, hence greatly shortening the computation time.[21]

To detect the feature envy code smell, Carneiro et al presented a multiple-view approach that enriches four categories of code views with concern properties, namely, concern's package-class-method structure, concern's inheritance-wise structure, concern dependency, and concern dependency weight.[22] In another research, van Emden and Moonen presented an approach for the automatic detection and visualization of several code smells including feature envy.[23] The approach performs automatic code inspection, relieving the developers of the manual inspection burden. Automatic inspection, reporting on the code's quality and conformance to coding standards, allows for the early (and repeated) detection of signs of project deterioration. Early feedback enables early corrections, thereby lowering the development costs and increasing the chances for success.

Fontana et al used various machine learning algorithms, such as J48, Naive, JRip, Random Forest, etc, to detect four code smells including feature envy and experimented their approach on 74 systems.[24] They showed that J48 and JRip algorithms performed better than other techniques in detecting feature envy code smells. They also concluded that the metrics ATFD (Access to Foreign Data), FDP (Foreign Data Providers), LAA (Locality of Attribute Accesses), NOA (Number of Attributes), and NMO (Number of Methods Overridden) play a significant role in the code smell detection approach. However, the approach is not able to refactor the code smell.

2.2 | Context-based refactorings

Contextual information, an important factor in OO design, adds a new dimension in the refactoring research field. The information provides the concept of a class that stands based on the SRP and, hence, a way of measuring cohesion and coupling of the application. Generally, IR techniques, such as latent semantic indexing and the vector space model (VSM), are used to capture the concept (or context) of a class.[25,26] Several literature works on this topic are discussed here.

Bavota et al proposed several approaches in order to identify chains of strongly related methods for *Extract Class* refactoring.[27-29] The approach analyzes both the structural and semantic similarities of the methods to group highly cohesive methods. The identified method chains are used to define new classes with higher cohesion than the original class. Moreover, the approaches used an advanced IR technique, namely, latent semantic indexing, for the refactorings. However, the approach is a semiautomated system because it takes as input a class previously identified by the software engineer as a candidate for the refactoring.

Ponisio and Nierstrasz proposed a group of contextual metrics that assess the cohesion of a package based on the degree to which its classes are used together by common clients.[30] The main idea is that if two classes of the package help fulfill the responsibility of a common client, they are conceptually related, regardless of the explicit relationships existing between them. In another paper, Gethers and Poshyvanyk proposed a coupling metric for OO software systems based on relational topic models, generative probabilistic model, to capture latent topics in source code classes and the relationships among them.[31] On the other hand, Poshyvanyk et al presented an approach for measuring coupling based on the conceptual information of classes.[32] The information is discovered from identifiers and comments of different classes related to each other. This type of relationship, called conceptual coupling, is measured through the use of IR techniques. The researchers claimed that the approach is different from existing coupling measures, because traditional approaches follow structural information, whereas this paper follows conceptual factors.

Several techniques were proposed to measure coupling among the code entities based on IR techniques. Although these work well to measure coupling among the classes, no analysis has been given for coupling between methods and classes. The techniques cannot recommend the proper class for refactoring the feature envy code smell due to not capturing method-level contextual information.

As stated above, these research studies address the importance of removing feature envy code smells through *move method* refactorings, as it occurs due to the violation of the two important design concepts: coupling and cohesion. Although various automated tools have been proposed throughout the years, all of those research studies have the common issue that these approaches are concerned only with the structural information (coupling and cohesion) of a source application rather than contextual information. Moreover, most of these research studies have considered only nonstatic entities (methods and attributes) and, hence, are not generalized for both static and nonstatic ones. Therefore, there is a lot more work needed in this research field to improve the *move method* refactoring approach.

3 | PROPOSED APPROACH

In this paper, a technique named "Move Method Refactoring Using Coupling, Cohesion, and Contextual Similarity" (**MMRUC3** framework) has been proposed to recommend *move method* refactoring. The technique uses contextual information along with dependency information for the recommendation. To analyze the source code robustly, it considers both static and nonstatic entities through analyzing entity names instead of reference names. The overall architecture of the MMRUC3 framework is shown in Figure 1.

According to the Figure, the technique first parses the source to find dependency and contextual information. Next, it calculates two types of similarity scores: dependency and contextual. These scores are then combined to suggest *move method* refactoring. In essence, the technique comprises four steps: *Source Code Analysis*, *Dependency (Coupling & Cohesion)-based Similarity Calculation*, *Contextual Similarity Calculation*, and *Move Method Refactoring Recommendation*. These steps are described as follows.

3.1 | Source code analysis

To refactor the feature envy code smell, methods and attributes in the source code of a software application are required to be analyzed. The reason is that these are the fundamental entities for measure coupling and cohesion. On the other hand, textual information from the source code is necessary to capture the context of the methods and classes. Hence, the technique analyzes the source code to obtain the essential factors such as method calls, used attributes, and contextual information of a method. These factors are used to calculate the following three metrics.

- *Coupling*—represents method calls and used attributes of other classes of the system by the caller method (a method that invokes other methods) of a class.

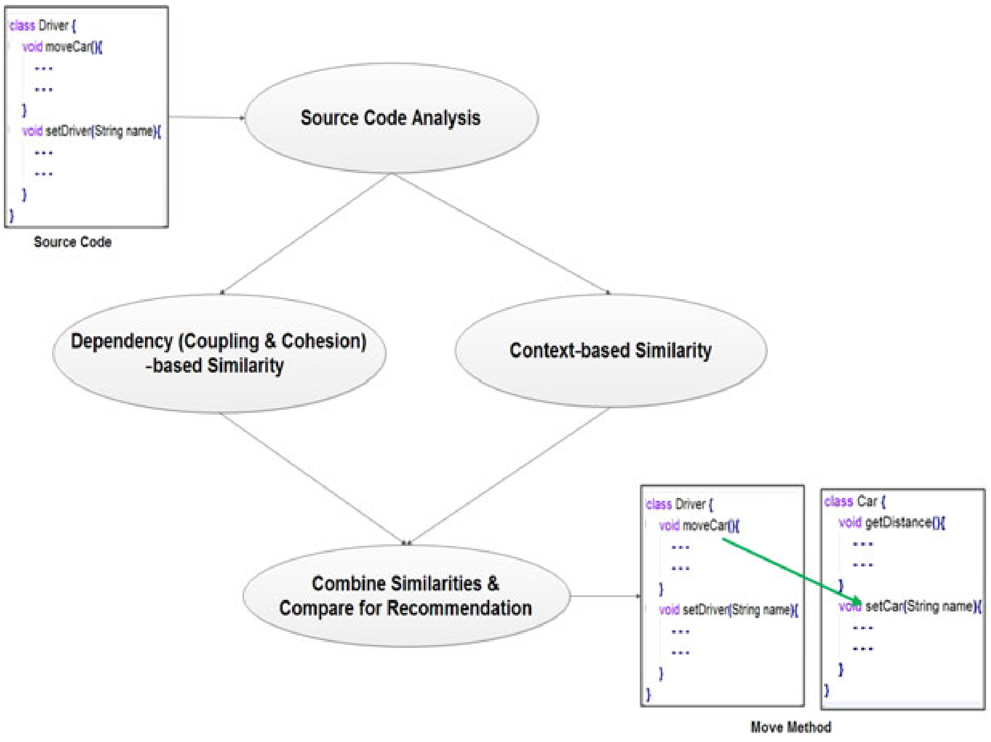

class Driver {
void moveCar(){
}
void setDriver(String name){
}
}
Source Code
Source Code Analysis
Dependency (Coupling & Cohesion) -based Similarity
Context-based Similarity
Combine Similarities & Compare for Recommendation
class Driver {
void moveCar(){
}
void setDriver(String name){
}
}
class Car {
void getDistance(){
}
void setCar(String name){
}
}
Move Method

FIGURE 1 Architecture of recommending the *move method* refactoring approach (MMRUC3 framework) [Colour figure can be viewed at wileyonlinelibrary.com]

- *Cohesion*—represents method calls and used attributes of the class in which the caller method belongs to.
- *Contextual factor*—The built-in keywords for the programming languages (used in the application) do not represent the context of a class. Therefore, the approach excludes these keywords and obtains textual information to achieve the more accurate context of the class.[33]

3.2 | *Move method* refactoring recommendation

Measuring similarities between methods and classes is significant in order to group similar behaviors (or methods) through the *move method* refactoring recommendation approach. After the source code parsing step, the MMRUC3 approach calculates the similarities using the parsed information of the *C3* factors: coupling, cohesion, and contextual information. The proposed recommendation approach is shown in Algorithm 1.

Algorithm 1. Recommendation of move method refactoring

Input: Target system S

Output: A list of candidate classes

```
1: for each method $m \in S$ do
2:     $C \leftarrow getClass(C)$
3:     $T \leftarrow \{\}$
4:     for each class $C_i \in S$ do
5:         if $jaccSim(m, C_i) + contextSim(m, C_i) > jaccSim(m, C) + contextSim(m, C)$ then
6:             $T \leftarrow T + \{C_i\}$
7:         end if
8:     end for
9:     $C' \leftarrow bestClass(m, T)$
10: end for
```

Assume that S is a system (source application) having a set of classes and m is a target method implemented in a class C of the system. For each method $m \in C$ (Line 1), information of the class C and the method m acquired from the parsing step is analyzed (Line 2). For each class $C_i \in S$ (Line 4), the algorithm determines whether m is more similar to the methods in C_i than to the methods in its current class C (Line 5). In fact, it computes two similarity scores using the $C3$ factors (coupling, cohesion, and contextual information) and combines the scores by mathematical addition. If C_i satisfies the condition of Line 5, that is, m is more similar with C_i (*Foreign Class*) than C (*Own Class*), then C_i will be a probable candidate class to receive method m. Such classes are inserted into a list (set) T (Line 6) initialized as an *empty set* in Line 3, as there can be multiple classes as candidates. Finally, the most suitable class C' 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 as parameters. It then sorts the candidate classes according to the similarity scores of the classes and provides the most appropriate class having the highest similarity score. Thus, the algorithm suggests *move method* refactoring in order to remove the feature envy code smell from the system S and eventually optimize modularization in terms of coupling and cohesion.

It is noticeable that the recommendation approach is an extension of the traditional *move method* recommendation approach. The traditional approach is to recommend a method to a class whose entities (methods and attributes) are used mostly by the method rather than due to similar behaviors. On the other hand, the proposed 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 and jointly implement a particular task or responsibility. For example, let us consider a method “sort” takes an array of integers and returns the sorted array. It calls two methods, ie, one method “compare” to compare two numbers and another method “swap” to swap two numbers. Hence, three methods “sort,” “compare,” and “swap” are implementing a particular feature, eg, “sorting.” From the viewpoint of the feature, these methods are jointly implementing the same task or responsibility.

The overall flowchart of the recommendation algorithm is shown in Figure 2. If the combined similarity score of the method's current class (*Own Class Similarity*) is less than other one or more classes (*Foreign Class Similarity*), then the

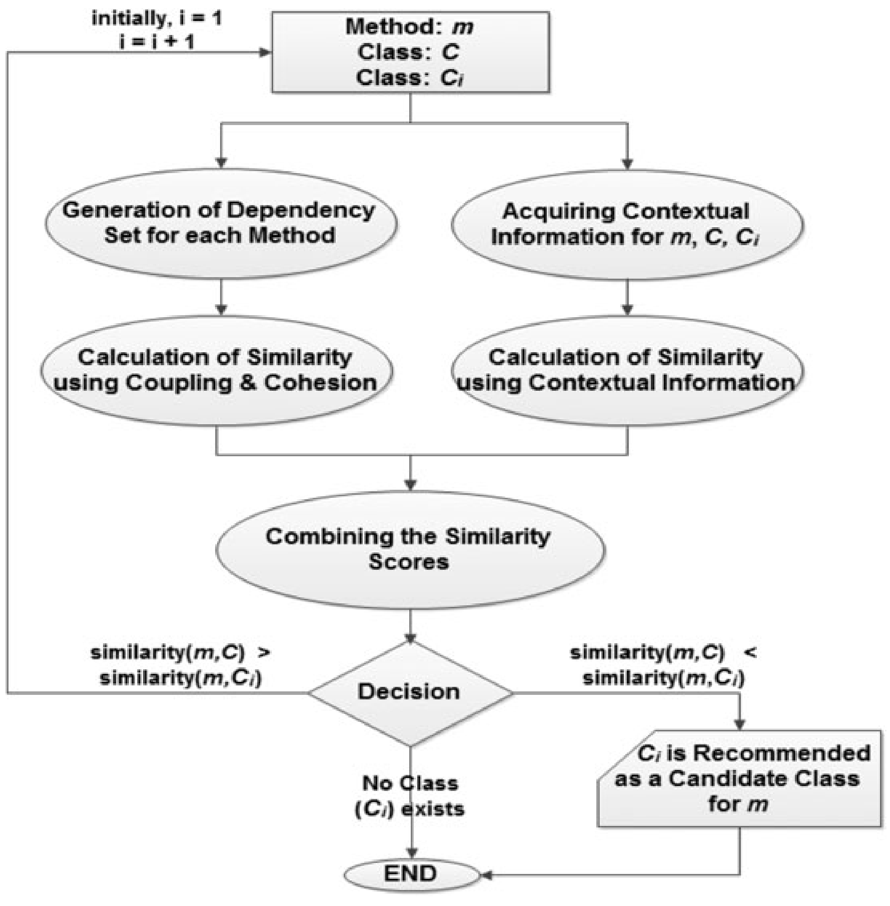

initially, i = 1
i = i + 1
Method: m
Class: C
Class: Ci
Generation of Dependency Set for each Method
Acquiring Contextual Information for m, C, Ci
Calculation of Similarity using Coupling & Cohesion
Calculation of Similarity using Contextual Information
Combining the Similarity Scores
similarity(m,C) > similarity(m,Ci)
Decision
similarity(m,C) < similarity(m,Ci)
Ci is Recommended as a Candidate Class for m
No Class (Ci) exists
END

FIGURE 2 Flowchart of the *move method* refactoring recommendation approach

technique detects the method as a feature envy code smell. In addition, it recommends the more appropriate class having the highest similarity score.

To measure the similarity score, the algorithm uses two similarity functions, as follows.

- *jaccSim()*—calculates similarity based on the dependencies, namely, coupling and cohesion (method calls and used attributes).
- *contextSim()*—calculates similarity using contextual information (class's and method's body information except for the application's built-in keywords).

Both of the key functions of the algorithm compute the similarities between the target method *m* and the methods in class *C*. The similarity functions incorporate both static and nonstatic entities and contextual information, respectively; those are the two main contributions of this approach. These functions are described briefly in the following subsections.

3.3 | Dependency (coupling & cohesion)-based similarity calculation

Coupling and cohesion are the two key design factors in any OO application. These design factors are essential for optimizing software modularization through decreasing coupling and increasing cohesion of the application. Therefore, similarity based on coupling and cohesion plays an indispensable role in the MMRUC3 recommendation approach.

In this similarity measurement step, dependency-based information such as method calls and used attributes by a target method, analyzed from the parsing step, is applied. The main task of this phase is to calculate the similarity between the target method and the classes based on the dependencies that represent coupling and cohesion. To measure the similarity, the Jaccard similarity coefficient is used in this step.[34] Jaccard similarity is a mathematical technique used for comparing similarity, dissimilarity, and distance of the data set. Measuring the Jaccard similarity coefficient between two data sets (*A* and *B*) is the result of division between the number of features that are common to both sets and the total number of features (Equation 1).

$$JaccardSimilarity(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{1}$$

The function relies on the set of structural dependencies (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. Dependency-based similarity function

Input: Target method *m* and a class *C*

Output: Similarity coefficient between *m* and *C*

```
1:  $similarityScore \leftarrow null$
2:  $avgSimilarityScore \leftarrow null$
3:  for each method $m_i \in C$ do
4:      if $m_i \neq m$ then
5:          $similarityScore \leftarrow similarityScore + getSimilarity(m, m_i)$
6:      end if
7:  end for
8:  if $m \in C$ then
9:      $avgSimilarityScore \leftarrow similarityScore/[NOM(C) - 1]$
10: else
11:     $avgSimilarityScore \leftarrow similarityScore/NOM(C)$
12: end if
13: return $avgSimilarityScore$
```

Initially, the function computes the similarity between the target method *m* and each other method m_i except *m* in class *C* (Line 5). It excludes the target method *m* by checking whether *m* and m_i are similar or not (Line 4) because it is illogical to measure similarity with the self. The cumulated similarity score for all methods in class *C* is assigned to a variable named *similarityScore*. In the end, the similarity between *m* and *C* is defined as the arithmetic mean of the similarity coefficients computed and assigned the score to the variable *avgSimilarityScore* in the previous step (Line 9 and Line 11).

Line 9 executes when the target method m belongs to class C, and Line 11 executes when m does not belong to C. In this algorithm, *NOM(C)* denotes the number of methods in a class C.

The key function of Algorithm 2 is *getSimilarity*(m, m_i), which computes the similarity between the sets of dependencies established by the two methods at Line 5. The similarity is measured by the use of the Jaccard similarity coefficient defined as Equation 2.

$$getSimilarity(m, m_i) = \frac{|A_m \cap A_{m_i}|}{|A_m \cup A_{m_i}|} \tag{2}$$

Here, A_m is the set of dependencies established by the target method m, and A_{m_i} is the set of dependencies established by other method m_i.

It is noted that the dependency set consists of the class names of the references or objects used for method calls and attribute usages. This dependency set is significant in the similarity measurement function, as it uses class names rather than reference names. It makes the approach more generic by incorporating both static and nonstatic entities (methods and attributes), whereas existing approaches have used reference names in the similarity calculation. Therefore, these existing approaches are not able to incorporate static entities, because objects cannot be created for using static entities in an OO system. Moreover, these static entities are used directly via class names of the entities. However, the inclusion of both static and nonstatic entities in the recommendation approach is a significant contribution of this article.

3.4 | Context-based similarity calculation

In this section, another significant kind of similarity is calculated, which is called contextual similarity. Similarity here refers that a method and a class are similar based on the context, as the class's responsibility (or context) is what its methods perform according to the SRP.[10] Hence, all the methods within the class should possess the same contextual information to enhance the cohesion of the application. The inclusion of similarity assists in achieving similar methods performing the single responsibility to be grouped into a class. Therefore, calculating similarity based on the contexts makes the recommendation approach more effective.

In this phase of the approach, contextual information is gained using IR techniques, such as *tf*, *idf*, and the *tf-idf* measure, and similarity is calculated using the technique of VSM, ie, *cosine similarity* on the basis of contextual information. The definitions and purposes of these terminologies are illustrated below.

- *tf*—refers to the number of occurrences of a *term* (t, unique word) in a document (D). It is denoted by $tf_{t,D}$ for a term t in the document D. For example, a document (D) has 1000 terms in total and a term t = *"football"* occurs 50 times in the document, D. Thus, $tf_{t,D} = 50$ for the document. However, the actual *tf* is gained by *logarithmic normalization*. Basically, *tf* is a property of a document. It is used to get the overall information based on the terms in a document. A method or class is referred to as a document in the approach.
- *Document frequency* (*df*)—refers to the number of documents that hold a term *(t)*. It is denoted by df_t for a term t. For example, there exist $N = 100$ documents in a collection and a term t = *"football"* appears in 10 documents in the collection. Thus, $df_t = 10$ for the collection. However, the actual *df* is gained by *logarithmic normalization*. Basically, *df* is a global property for a term in the collection.
- *idf*—is used to discriminate each document based on the uniqueness of a term. It is also a global property of a term in the document collection, defined as

$$idf_t = \log_2(N/df_t).$$

- *tf-idf*—combines the definitions of *tf* and *idf* to produce a composite weight for each term in each document. The *tf-idf* weighting scheme assigns to term t a weight in document D given by

$$tf - idf_{t,D} = tf_{t,D} * idf_t.$$

  It is used to produce a composite weight for each term in each document. It helps distinguish the responsibilities among various classes of an application.
- *Cosine similarity*—is used to calculate the similarity score between two documents, namely, a method and a class.

The function relies on the context that a target method and a class stand for accomplishing a specific task (or responsibility). The context is determined from the texts of the method's and the class's bodies (referred to as documents), described in Algorithm 3. The algorithm takes a method *m* and a class *C* as input, calculates the contextual similarity between them, and returns the calculated score as output.

Algorithm 3. Contextual similarity function

```
Input: Target method $m$ and class $C$
Output: Similarity between $m$ and $C$
1: Index used fields and invoked methods in method $m$
2: Index declared fields and methods of class $C$
3: Calculate term frequency tf (number of occurrences) of each indexed elements of $m$ & $C$
4: Calculate inverse document frequency idf of each indexed elements of $m$ & $C$
5: Calculate tf-idf of each indexed elements of $m$ & $C$ using the formula:
   tf-idf of element, $e \leftarrow (idf * \log 2(1 + tf))/max\_frequency(e)$
6: Calculate cosine similarity between tf-idf vectors of $m$ & $C$ using the formula:
   $Cosine(m, C) \leftarrow (vector(m).vector(C))/(|vector(m)| * |vector(C)|)$,
   Where $vector(m).vector(C)$ = dot product of $vector(m)$ and $vector(C)$, $|vector(C)|$ = magnitude
   of vector $C = \sum \sqrt{(Ci^2)}$
7: Return Cosine Similarity Value
```

At first, the body of method *m* is parsed and tokenized to identify the used fields, declared variables, and method invocations. After that, such information is stored in a vector. The vectors can be regarded as bags according to the *bag-of-words model*, since the vector does not consider the position of the words (or items). Similarly, the body of class *C* is parsed to identify declared fields and declared methods, and such information is also stored in another vector. Then, the number of occurrences of each unique element in the vectors is calculated, which is known as *tf* (Line 3). In fact, *tf* is the *logarithmic normalization* (*normalized* $tf_t = 1 + \log_2(tf_t)$) form to ensure the significance of an element. In the rest of the cases, *tf* refers to the *normalized tf*. After calculating the *tf*, *idf* is calculated for each unique element of the vectors. The *idf* is a numerical statistic that is intended to reflect how important a word (or term) is to a document in a collection or corpus. The formula to calculate *idf* for a term (t) in a document (D) is given as

$$idf_{t,D} = \log_2 \frac{N}{|\{d \in D : t \in d\}|}, \tag{3}$$

where N is the number of documents in the collection, and $|\{d \in D : t \in d\}|$ is the number of documents where the term t occurs.

After calculating the *idf* of each unique element, the *tf-idf* score of each element in the vectors is calculated in Line 5 using Equation 4 and stored in separate *tf-idf* vectors. Using the *tf-idf* vectors of method *m* and class *C*, the cosine similarity score between those vectors is calculated using Equation 5 in Line 6. Finally, the calculated cosine similarity score is returned.

$$tf - idf_{t,D} = tf_{t,D} * idf_{t,D} \tag{4}$$

$$CosineSimilarity(d1, d2) = \frac{\vec{V}(d1).\vec{V}(d2)}{|\vec{V}(d1)||\vec{V}(d2)|} \tag{5}$$

Here, Equation 5 uses the VSM of the IR technique to calculate the similarity between the target method and a class represented as two documents *d1* and *d2*. It computes the cosine similarity of their vector representations $\vec{V}(d1)$ and $\vec{V}(d1)$, where the numerator represents the dot product (also known as the inner product) of the vectors $\vec{V}(d1)$ and $\vec{V}(d2)$, whereas the denominator is the product of their Euclidean lengths. The overview of the similarity measurement process is shown in Figure 3.

It is pointed out that the proposed approach uses similarity based on not only the coupling and cohesion but also the contextual information of a method and a class. Contexts are analyzed from the method's and the class's bodies. Therefore, the application should follow the appropriate coding conventions, such as the naming convention of methods, classes, attributes, etc, in order to get effective results of the approach.

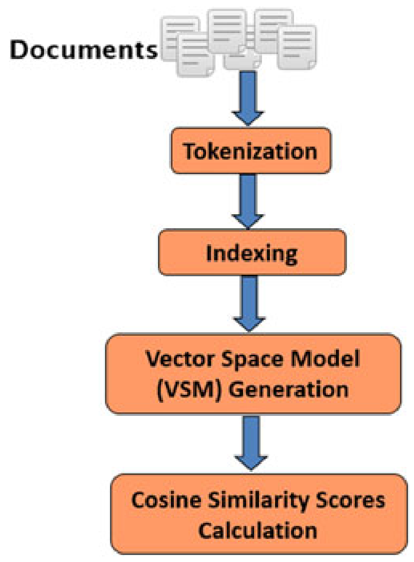

Documents
Tokenization
Indexing
Vector Space Model (VSM) Generation
Cosine Similarity Scores Calculation

FIGURE 3 Schematic diagram of the contextual similarity measurement process [Colour figure can be viewed at wileyonlinelibrary.com]

3.5 | Complexity analysis of the MMRUC3 algorithm

The complexity of an algorithm is the function *f*(*n*), which gives the running time and/or storage space requirement of the algorithm in terms of the size *n* of the input data.[35] An algorithm has mainly two types of complexity.

- *Time complexity*—how fast the algorithm runs.
- *Space complexity*—how much space is needed for executing the algorithm.

Time complexity is the concern of the refactoring research field, as the target of refactoring is to make faster the development and maintenance activities. Therefore, this subsection analyzes only the *time complexity* of the MMRUC3 algorithm. The complexity is based on the numbers of fields, methods, and classes of an input project.

Assumptions for the complexity computation:
m_i = number of methods in a class c_i
f_j = number of fields in a method m_j
x = indexing time for a field
$f_i = \sum(m_j * f_j)$ = total number of fields in a class c_i
c = total number of classes in the project
$f = c * \sum(m_j * f_j)$ = total number of fields in the project
Hence, complexity = indexing time + similarity calculation time $= (f * x) + c * \sum(m_j * f_j) * f = (f * x) + (c * f_i * f)$

4 | CASE STUDY ON A SAMPLE PROJECT: "VideoStore"

In this section, a detailed walk-through on the proposed recommendation approach is given as a case study. For this, a project named *VideoStore* is selected, because the project possesses all the *C3* factors (coupling, cohesion, and context) required to calculate the similarities between methods and classes, as mentioned in Section 4.

4.1 | About project *VideoStore*

The project *VideoStore* is a well-known example of refactoring usage mentioned in the book *Refactoring: Improving the Design of Existing Code*.[1] The project is well designed and self-descriptive, as it follows proper OO naming conventions and *State Design Pattern*.[36] That is why the project is used as the case study in this article, so that it will be easier to visualize the proposed MMRUC3 recommendation approach properly.

This project calculates and outputs a statement of a customer's charges at a video store based on the renting movie types (regular movie, children's movie, or new release movie) and duration of renting. In addition to calculating charges,

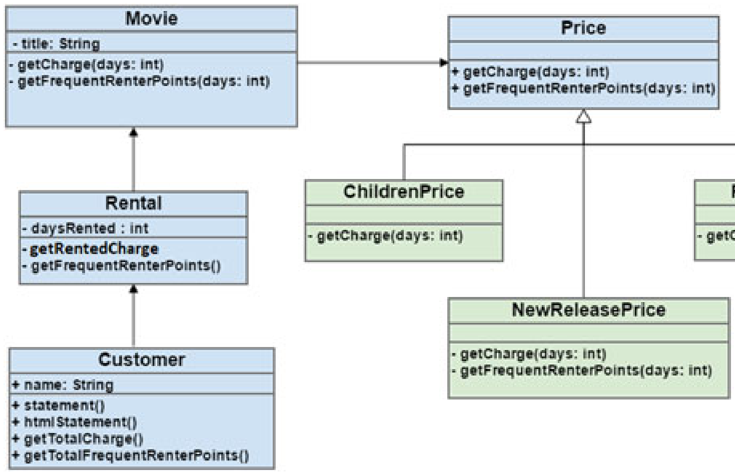


FIGURE 4 UML diagram of the *VideoStore* project (refactored version) [Colour figure can be viewed at wileyonlinelibrary.com]

it also computes frequent renting points, which vary depending on whether the movie is a new release or not. The UML diagram of the project is shown in Figure 4.

The *VideoStore* project consists of seven classes: *Movie*, *Price*, *RegularPrice*, *ChildrenPrice*, *NewReleasePrice*, *Rental*, and *Customer*. Each class has a responsibility to perform depending on the renting movie types. The responsibilities of each class are described below.

- **Class** *Movie*: The responsibility of the class is to determine the movie type and the price for renting the movie by a customer. In addition, it also counts points for a specific movie type.
- **Class** *Price*: This is an abstract class developed using the polymorphism property of the OO approach. The main functionality of the class is to calculate the price for each type of movies (Regular, Children, and New Release) based on the number of rented days. To perform the function, this super class has three subclasses (*RegularPrice*, *ChildrenPrice*, and *NewReleasePrice*) based on movie types. It also holds the inheritance relationship between the super classes and subclasses.

  - ☐ **Subclass** *RegularPrice*: It inherits the class *Price* and calculates the price for regular-type movies.
  - ☐ **Subclass** *ChildrenPrice*: It inherits the class *Price* and calculates the price for children-type movies.
  - ☐ **Subclass** *NewReleasePrice*: It inherits the class *Price* and calculates the price for new release–type movies. It also performs calculating points for renting the movies.

- **Class** *Rental*: Customers use the class for renting any types of movie. Based on the customer's requirements, this class calculates renting information such as movie type and price using the *Movie* class.
- **Class** *Customer*: This is a client class by which renting information is viewed after a rent request is made.

However, for the step-by-step procedure of the MMRUC3 framework, the *VideoStore* project is modified by extracting the *getRentedCharge*() (red colored) method from the *Rental* class to the *Movie* class as well as the *getPrice(int days)* (red colored) method from the *Price* class to the *Movie* class, that is, injecting feature envy code smells manually, as shown in Figures 5A and 5B, respectively. Moreover, placing the methods (target methods) into inappropriate classes increases the interactions (high coupling) shown by red arrows in the Figure and responsibilities of the classes (low cohesion). Now, the MMRUC3 proposed approach can be applied on the nonrefactored version of the project to recommend *move method* refactorings to optimize these design factors.

4.2 | Recommending *move method* refactorings for *VideoStore*

For each feature envy code smell, the MMRUC3 approach is executed on the modified version of the *VideoStore* project. After parsing the source classes of the project, similarities are calculated between the methods and classes using the analyzed information. Based on the similarity scores, *move method* refactorings are recommended. For the purpose of simplicity, the *Customer* class and the subclasses of the *Price* class are excluded from the computation, because the first

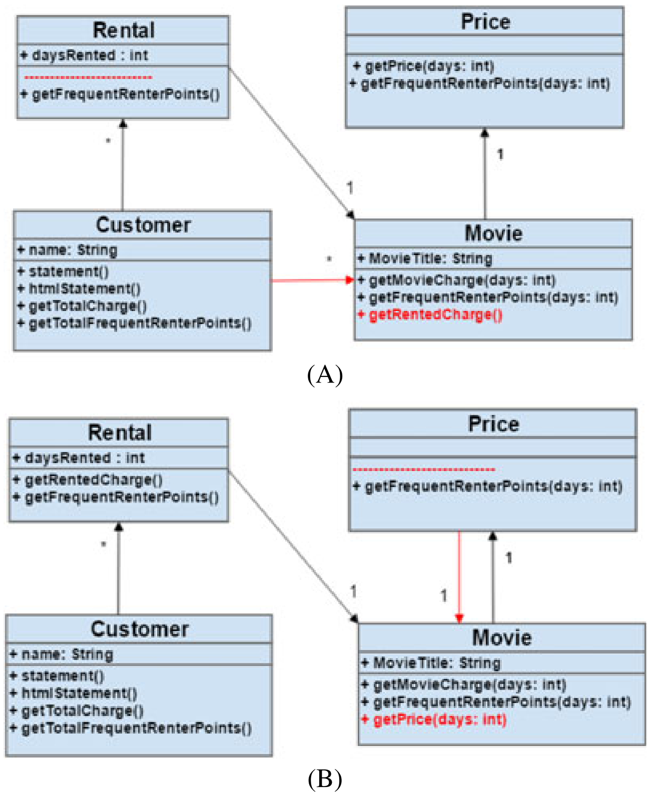


FIGURE 5 UML diagram of the *VideoStore* project (nonrefactored version). A, Feature envy code smell injected for method *getRentalCharge*(); B, Feature envy code smell injected for method *getPrice(int days)* [Colour figure can be viewed at wileyonlinelibrary.com]

one is the client class developed for a different context, and the subclasses' context is covered by its super class. The main phases of the case study are described briefly in this subsection, as follows.

1. Parsing & Analyzing *VideoStore* Source Project
2. Similarity Coefficient Measurement
3. Recommendation of *Move Method* Refactorings

4.2.1 | Parsing & analyzing the *VideoStore* project

In order to acquire significant information through parsing, several preprocessings are made in the *VideoStore* project before analyzing the source code information. First of all, classes (*.java* files) of the project are compiled to get the byte codes (*.class* files) of the corresponding classes. The *.class* files are not in readable form, and therefore, these files are converted into text forms (*.txt* files) using the following command.

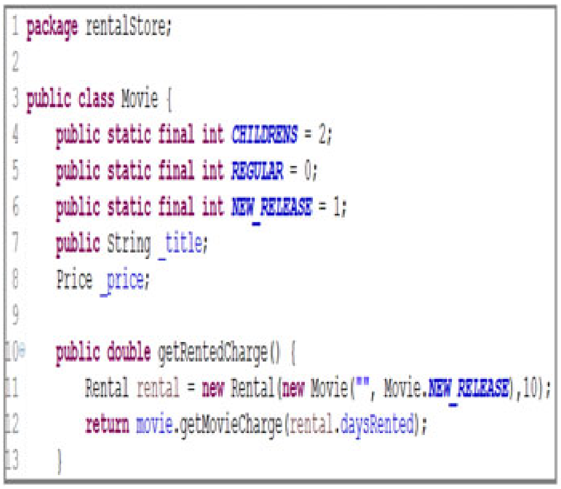
"javap -c -private ClassName.class > ClassName.txt"

As for example, Figures 6A and 6B show the source code portion of *Movie.java* and the byte code portion (*Movie.class*) of that class after the conversion, respectively.

After the conversion, those byte and base source codes have been considered in the form to be parsed. In sum, both *.class* files (or *.txt* files) and *.java* files are used for parsing the source information from the *VideoStore* project. An intelligent parser, made from the combination of the *ByteParser*[37] and the *JavaParser*,[38] is used to parse the source files. Those files have then been parsed to get all the classes, namely, *Customer, Rental, Movie, Price, ChildrenPrice, NewReleasePrice, and RegularPrice*, and all methods in each class of the project.

After parsing, information is acquired from method invocations, attribute usages, and context of the source codes. The information is based on the dependency and context of the source classes of the project.

From Figure 5A, it is seen that the method *getRentedCharge*() is placed into the inappropriate class *Movie* from the appropriate one *Rental*. Therefore, coupling increases the application. The codes of these classes are shown in Figure 7.

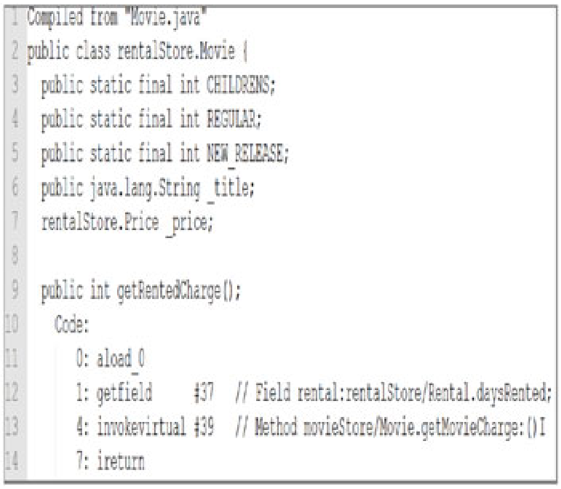


```
package rentalStore;

public class Movie {
    public static final int CHILDRENS = 2;
    public static final int REGULAR = 0;
    public static final int NEW_RELEASE = 1;
    public String _title;
    Price _price;

    public double getRentedCharge() {
        Rental rental = new Rental(new Movie("", Movie.NEW_RELEASE),10);
        return movie.getMovieCharge(rental.daysRented);
    }
```

(A)

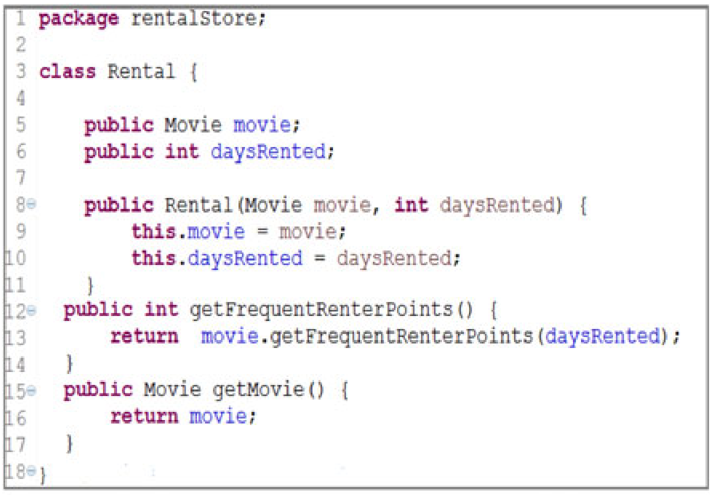


```
Compiled from "Movie.java"
public class rentalStore.Movie {
  public static final int CHILDRENS;
  public static final int REGULAR;
  public static final int NEW_RELEASE;
  public java.lang.String _title;
  rentalStore.Price _price;

  public int getRentedCharge();
    Code:
      0: aload_0
      1: getfield      #37   // Field rental:rentalStore/Rental.daysRented;
      4: invokevirtual #39   // Method movieStore/Movie.getMovieCharge:()I
      7: ireturn
```

(B)

FIGURE 6 Examples of the Java and Byte codes. A, Source code example of the *Movie* class; B, Byte code example of the *Movie* class [Colour figure can be viewed at wileyonlinelibrary.com]



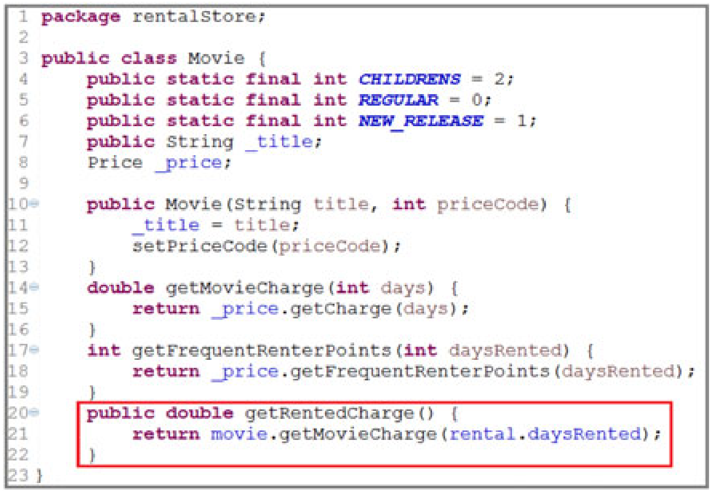
```
package rentalStore;

class Rental {

    public Movie movie;
    public int daysRented;

    public Rental(Movie movie, int daysRented) {
        this.movie = movie;
        this.daysRented = daysRented;
    }
  public int getFrequentRenterPoints() {
      return  movie.getFrequentRenterPoints(daysRented);
  }
  public Movie getMovie() {
      return movie;
  }
}
```

(A)



```
package rentalStore;

public class Movie {
    public static final int CHILDRENS = 2;
    public static final int REGULAR = 0;
    public static final int NEW_RELEASE = 1;
    public String _title;
    Price _price;

    public Movie(String title, int priceCode) {
        _title = title;
        setPriceCode(priceCode);
    }
    double getMovieCharge(int days) {
        return _price.getCharge(days);
    }
    int getFrequentRenterPoints(int daysRented) {
        return _price.getFrequentRenterPoints(daysRented);
    }
    public double getRentedCharge() {
        return movie.getMovieCharge(rental.daysRented);
    }
}
```

(B)

FIGURE 7 Source code examples of *VideoStore* (target smelly method *getRentedCharge*()). A, Class *Rental*; B, Class *Movie* [Colour figure can be viewed at wileyonlinelibrary.com]

Dependency-based information

The body of the target method *getRentedCharge*() is analyzed to gain dependency-based information. This information is called the *Dependency Set* of the method. Here, the formal method notation[39] *Set* is used because it does not hold redundant information. The Dependency Set contains the class names by which the method uses features (other methods and attributes), given as follows.

$$DependencySet_{getRentedCharge()} = \{Movie, Rental\}$$

Similarly, dependency sets for the system classes *Movie* excluding the target method (*Own Class* containing the target method), *Rental*, and *Price* are captured. Class-level dependencies are calculated based on all methods inside a class. The dependency sets are represented as follows.

$$DependencySet_{Move} = \{Movie, Price\}$$
$$DependencySet_{Rental} = \{Movie\}$$
$$DependencySet_{Price} = \{\}$$

Contextual information

Another significant information is analyzed from the context of the target method *getRentedCharge*(). This contextual information is called the *Contextual Bag* of the method. Here, the formal method notation[39] *Bag* is used because it can hold redundant information and does not consider the position of words according to the *bag-of-words* model.[12,26] The Contextual Bag contains words of the method's body including its name.

$$ContextualBag_{getRentedCharge()} = [\![get, rented, charge, movie, get, movie, charge, rental, days, rented]\!]$$

Similarly, contextual bags for the system classes *Movie* excluding the target method (*Own Class* containing the target method), *Rental*, and *Price* are captured. The contextual bags are represented as follows.

$$ContextualBag_{Movie} = [\![movie, children, regular, new, release, title, price, movie, title, price, code, title, title, set, price, code, price, code, get, movie, charge, days, price, get, charge, days, get, frequent, renter, points, days, rented, price, get, frequent, renter, points, days, rented, get, rented, charge, movie, get, movie, charge, rental, days, rented]\!]$$
$$ContextualBag_{Rental} = [\![rental, movie, movie, days, rented, rental, movie, movie, days, rented, movie, movie, days, rented, days, rented, get, frequent, renter, points, movie, get, frequent, renter, points, days, rented, movie, get, movie, movie]\!]$$
$$ContextualBag_{Price} = [\![price, get, price, code, get, charge, days, rented, days, get, frequent, renter, points, rented]\!]$$

Note that such contextual information excludes the *java* programming language keywords and is tokenized into multiple words according to camel case, pascal case, white spaces, etc, for each word of the methods and classes. These bags of words are converted into lowercase letters for removing ambiguity.

4.2.2 | Similarity coefficient measurement

After analyzing the method- and class-level information of the *VideoStore* project, the MMRUC3 framework measures two types of similarity, namely, dependency (coupling & cohesion)-based similarity and context-based similarity, which are described below.

Dependency (coupling & cohesion)-based similarity: The coupling- and cohesion-based similarity is calculated using the dependency sets of the target method and all classes of the project. Here, the *Jaccard similarity coefficient* formula (Equation 1) is used.

$$JS1(m, Movie) = \frac{|DependencySet_m \cap DependencySet_{Movie}|}{|DependencySet_m \cup DependencySet_{Movie}|} = \frac{|\{Movie\}|}{|\{Movie, Rental, Price\}|} = \frac{1}{3} = 0.33$$
$$JS2(m, Rental) = \frac{|DependencySet_m \cap DependencySet_{Rental}|}{|DependencySet_m \cup DependencySet_{Rental}|} = \frac{|\{Movie\}|}{|\{Movie, Rental\}|} = \frac{1}{2} = 0.5$$

Here, m is the target method *getRentedCharge*(), $JS1$ is the Jaccard similarity score for the first class *Movie*, and $JS2$ is the Jaccard similarity score for the second class *Rental*.

TABLE 1 Mathematically cosine similarity calculation between method *getRentedCharge*() and class *Movie*

Term t_i	idf_i $\log_2(N/df_{t_i})$	D_m for m = "getRentedCharge()" tf	Normalized tf_i $1+\log_2(t_i)$	$tf\text{-}idf_i$	D_1 for C_1 = "Movie" tf_i	Normalized tf_i $1+\log_2(t_i)$	$tf\text{-}idf_i$	Similarity Score (m,C_1)
get	0	1	1	0	6	3.58	0	
rented	0	2	2	0	4	3	0	
charge	0	2	2	0	4	3	0	0.9899
movie	0.58	2	2	1.16	5	3.32	1.74	
rental	0.58	1	1	0.58	1	1	0.58	
days	0	1	1	0	5	3.32	0	

TABLE 2 Mathematically cosine similarity calculation between method *getRentedCharge*() and class *Rental*

Term t_i	idf_i $\log_2(N/df_{t_i})$	D_m for m = "getRentedCharge()" tf	Normalized tf_i $1+\log_2(t_i)$	$tf\text{-}idf_i$	D_2 for C_2 = "Rental" tf_i	Normalized tf_i $1+\log_2(t_i)$	$tf\text{-}idf_i$	Similarity Score (m,C_2)
get	0	1	1	0	3	2.58	0	
rented	0	2	2	0	5	3.32	0	
charge	0	2	2	0	0	0	0	0.969
movie	0.58	2	2	1.16	10	4.32	2.66	
rental	0.58	1	1	0.58	1	1	0.58	
days	0	1	1	0	5	3.32	0	

TABLE 3 Mathematically cosine similarity calculation between method *getRentedCharge*() and class *Price*

Term t_i	idf_i $\log_2(N/df_{t_i})$	D_m for m = "getRentedCharge()" tf	Normalized tf_i $1+\log_2(t_i)$	$tf\text{-}idf_i$	D_3 for C_3 = "Price" tf_i	Normalized tf_i $1+\log_2(t_i)$	$tf\text{-}idf_i$	Similarity Score (m, C_3)
get	0	1	1	0	3	2.58	0	
rented	0	2	2	0	1	1	0	0
charge	0	2	2	0	1	1	0	
days	0	1	1	0	2	1	0	

Contextual similarity: The context-based similarity is calculated using the contextual bags (also known as documents in *IR*) of the target method and all classes of the project. Here, the *cosine similarity* formula is used as in Equation 3. To measure cosine similarity, the documents are considered as vectors according to the VSM, and so, the length of the two documents should be the same.[26] To do that, the terms of the method are considered as the vector dimensions. The step-by-step similarity calculation process is shown in Tables 1, 2, and 3 for the target method *getRentedCharge*() and the three classes (*Movie, Rental, and Price*) of the projects, respectively.

Meaning of the symbols used in Tables 1-5:
m = the target method *getRentedCharge*();
$i = \{1, 2, 3, \ldots\}$;
C_1 = class *Movie*, C_2 = class *Rental*, C_3 = class *Price*;
D_i = document for class C_i, D_m = document for method m;
$tf\text{-}idf_i = tf_i * idf_i$;
JSi = Jaccard similarity score between m and C_i;
CSi = cosine similarity score between m and C_i;
TSi = total similarity score between m and C_i.

Combining the two types of similarity: In this step, the two similarities are combined by mathematical addition in order to gain the more accurate and total similarity scores between the target method and the classes. The overall similarity scores between the method *getRentedCharge*() and all the classes of the project are shown in Table 4.

TABLE 4 Similarity scores for all classes for method *getRentedCharge*()

Jaccard Similarity JSi		Contextual Similarity CSi		Total Similarity Scores TSi = (JSi + CSi)	
JS1	0.333	CS1	0.989	TS1	1.332
JS2	0.5	CS2	0.969	TS2	1.469
JS3	0	CS3	0	TS3	0

TABLE 5 Similarity scores for all classes for method *getPrice*()

Jaccard Similarity JSi		Contextual Similarity CSi		Total Similarity Scores TSi = (JSi + CSi)	
JS1	0.333	CS1	0.945	TS1	0.945
JS2	0.5	CS2	0	TS2	0
JS3	0	CS3	1	TS3	1

TABLE 6 Recommendation of *move method* refactorings

Smelly Method	Current Class	Recommended Class
getRentedCharge()	Movie	Rental
getPrice(int days)	Movie	Price

The similarity scores between another smelly target method *getPrice*() (shown in Figure 11B) and all the three classes of the *VideoStore* project are shown in Table 5 using the same process as described above.

4.2.3 | Recommendation of *move method* refactorings

From Table 4, it is seen that TS2, that is, the total similarity score between the target method *getRentedCharge*() and class *Rental*, gets the highest combined similarity score and, eventually, is higher than the Own Class *Movie* in which the method is kept. As a result, the method is placed in the inappropriate class, and hence, it is detected as the feature envy code smell. Therefore, comparing the scores, the MMRUC3 framework recommends the *move method* refactoring for the method, that is, the method should be moved to class *Rental* from its current class *Movie*. Similarly, it recommends the *Price* class for another smelly method *getPrice*(). The final recommendations of *move method* refactorings for the two smelly methods (Figure 5) of the *VideoStore* project are shown in Table 6. These refactorings make the project achieve the original refactored project as shown in Figure 4, which ensure loose coupling and high cohesion.

It is noted that CS2 is less than CS1, whereas JS2 is greater than JS1 in Table 4. However, after combining the two similarity scores, TS2 gets a higher score than TS1. From these findings, it is stated that only the cosine similarity (context based) cannot perform the recommendation correctly. On the other hand, for the other injected feature envy code smell, it is shown that only the *Jaccard* similarity (dependency based) cannot perform the recommendation appropriately, as shown in Table 5. Therefore, the final statement of these findings is that some methods can be suggested as refactoring candidates using dependency-based similarity, whereas contextual similarity fails, and vice versa. However, the combination of the two similarities shows more accuracy of the recommendation approach.

5 | EXPERIMENTAL RESULT ANALYSIS

To assess the proposed approach, preliminary experiments have been conducted by developing a prototype of the proposed MMRUC3 algorithm in Java language. The existing widely used refactoring tool, ie, JDeodorant, has also been used for comparative analysis.

5.1 | Experimental setup

In order to perform the experimentation, the prototype has been developed in Java. An open source parser named ByteParser[37] has been used to parse essential factors from the source code. ByteParser is an extension of JavaParser[38] that supports byte code analysis. The reason for analyzing the byte code is that it ensures that the source code is compilable

TABLE 7 Experimental projects

Id No.	Project	Version	NOC	NOM	LOC
1	JHotDraw	7.6	674	6533	80 536
2	ArgoUML	0.34	1291	8077	67 514
3	JMeter	2.5.1	940	7990	94 778
4	FreeMind	0.9.0	658	4885	52 757
5	Maven	3.0.5	647	4888	65 685
6	DrJava	r583	788	7156	89 477
7	Weka	3.6.9	1535	17 851	272 611
8	VideoStore	1.0	7	25	212

Abbreviations: LOC, line of codes; NOC, number of classes; NOM, number of methods.

and refactoring should be carried out on the executable and compilable code.[1] Besides, the parser has also been used earlier to detect a code smell named *Dead Field* in a test code.[40]

For the validation of the approach, eight open source Java projects have been used as data sets. These projects, used by existing research studies, have been collected from the online repository.[13] The descriptions of the projects are shown in Table 7, consisting of five columns. The columns of the table represent the project id, project name, project version, NOCs, NOMs, and LOCs, respectively. Each project has a large number of NOCs, NOMs, and LOCs, except the last one. In fact, the last one, ie, *VideoStore*, used as examples of code smells and refactorings, has been collected from the book *Refactoring: Improving the Design of Existing Code*.[1] Since the project is standard and organized, it has been selected as the data set and case study (Section 4: Case Study). From the table, it is seen that *Weka* is the largest project, and *Maven* and *FreeMind* are the smallest ones on the basis of NOCs, NOMs, an LOCs, except *VideoStore*.

The abovementioned eight projects have been used for evaluating the MMRUC3 approach. The obtained experimental results are discussed in the next subsections.

5.2 | Result analysis

The quality of a recommendation system is typically measured using the *Precision* and *Recall* metrics of IR techniques.[12] The evaluations based on the metrics are discussed here.

These metrics are calculated based on the following factors.

- *True Positive, TP*—the number of affected instances (ie, methods) that are recommended correctly. Affected instances mean misplaced methods.
- *False Positive, FP*—the number of instances that are recommended incorrectly.
- *False Negative, FN*—the number of affected instances that are not recommended by the approach.
- *True Negative, TN*—the number of nonaffected instances that are not recommended by the approach.

These notions can be made clear by examining the following contingency table (Table 8).

Precision: Precision measures how well the recommender approach filters out the incorrect results. It is the fraction of the returned correct results in the overall result set. Thus, the fraction of the true positive results in the total returned results (true positive + false positive) is called precision (Equation 6).

$$Precision = \frac{TP}{TP + FP} \tag{6}$$

TABLE 8 Contingency table for the result analysis of MMRUC3

	Affected Methods	Nonaffected Methods
Recommended	True Positive (TP)	False Positive (FP)
Nonrecommended	False Negative (FN)	True Negative (TN)

TABLE 9 Results of the proposed MMRUC3 approach

Id No.	Project	TP #	FP #	FN #	Precision, %	Recall, %
1	JHotDraw	17	52	3	24.64	85.00
2	ArgoUML	24	30	7	44.44	77.42
3	JMeter	7	36	3	16.28	70.00
4	FreeMind	6	36	4	14.29	60.00
5	Maven	14	52	9	21.21	60.87
6	DrJava	7	144	2	4.64	77.78
7	Weka	14	189	10	6.90	58.33
8	VideoStore	2	0	0	100.00	100.00
	Average				***29.05***	***73.68***

Abbreviations: FN, False Negative; FP, False Positive; TP, True Positive.

Recall: Recall measures how well the recommender finds the correct results. It is the fraction of the correct returned results in the overall collection of correct results. Thus, the fraction of the true positive results with the total relevant results (true positive + false negative) is called recall (Equation 7).

$$Recall = \frac{TP}{TP + FN} \tag{7}$$

The results of the proposed MMRUC3 approach are shown in Table 9. The table columns are project id, project name, TP, FP, FN, Precision, and Recall.

The precision and recall metrics are calculated using Equations 6 and 7. For instance, assume that for the *JHotDraw* project, $TP = 17$, $FP = 52$, and $FN = 3$. Therefore, the precision and recall of the project are as follows.

$$Precision = \frac{TP}{TP + FP} = \frac{17}{17 + 52} = 0.2464 = 24.64\%$$

$$Recall = \frac{TP}{TP + FN} = \frac{17}{17 + 3} = 0.85 = 85\%$$

It is observed that the approach gets the highest precision (100%) and recall (100%) for the *VideoStore* project. The project size is very small compared to the others in terms of NOC, NOM, and LOC. In addition, it is a well-standard project, as it has followed proper naming conventions of OO design.[1] Therefore, the MMRUC3 approach efficiently acquires both the dependency- and context-based information of the project for the recommendations.

The second highest precision of 44.44% is for the large project *ArgoUML* and the lowest precision of 6.90% is with the *Weka* project. Overall, for the large projects, *ArgoUML*, *JHotDraw*, and *Maven* have higher precisions (above 20%); *JMeter* and *FreeMind* have moderate ones (between 10% and 19%); and finally, *Weka* and *DrJava* get lower ones (below 10%). Similarly, *JHotDraw*, *DrJava*, and *ArgoUML* have higher recalls (above 75%); *JMeter* and *FreeMind* have moderate ones (between 65% and 74%); and finally, *Maven*, *FreeMind*, and *Weka* get lower ones (below 65%). The results vary from project to project because of the projects' nature (size and coding standard). It is also seen from the table that the project *DrJava* gets lower precision but moderate recall because the project has a moderate number of classes having a lot of methods and statements (Table 1). Similarly, *Weka* gets lower precision due to its exalted size. Because of the higher size, there is a possibility of getting false positive results.

For avoiding the biasness of a small project, *VideoStore* is excluded from the further calculation of the average precision and recall of the approach. Therefore, the average precision is 18.91% and the recall is 69.91% after the exclusion. The subsequent subsections of the result analysis also exclude this small project.

Since each project has been developed following different coding standards (such as naming conventions, coding sizes, etc) and the approach deals with the contexts, the results thus depend on the projects' standards. The variations in the results indicate that there exist relationships between the approach and the projects. These relationships are discussed in the next subsection.

5.3 | Relationships between MMRUC3 and the projects

In the previous subsection, it is stated that the accuracy of the MMRUC3 approach varies on the basis of the project nature (project standards and sizes). Therefore, there exist some patterns or relationships between the approach and the projects. These relationships are established in this subsection.

5.3.1 | Relationship based on project standards

An important code quality aspect of large-scale software development is conformance to coding standards. Coding standards ensure that everyone in a software company can understand the codes and work with each other. If the conformance is not achieved (ie, if the code is not written and organized according to the programming guidelines), it becomes much harder for a large team of programmers to develop, integrate, and maintain a particular piece of software and find errors.[23,41,42] Therefore, concise and consistent naming conventions should be followed to improve readability and comprehensibility of the software application.[42]

In order to discover the relationship between the MMRUC3 approach and the project based on the coding standard, it is necessary to define the *project standard*. As the approach considers contextual factors, *project standard* is defined by the naming conventions of a project. The more the project is readable, comprehensible, and self-descriptive, the higher the *project standard* is.

The categories (and priority values) of the *project standard* are given below.

- **1 for Excellent:** All classes of a project follow the exact naming convention that is easily understandable and self-descriptive. For instance, the name of the method *calculateSalary* is comprehensible and self-descriptive.
- **2 for Best:** Most of the classes of a project are named properly, but few names are not self-descriptive.
- **3 for Better:** Almost all classes of a project follow appropriate name conventions. However, few names should be completed to be easily understandable, such as *calSalary* should be *calculateSalary*.
- **4 for Good:** The naming convention is followed properly. However, some names are confusing to understand such as *cSalary*.
- **5 for others (nonstandard):** The projects that do not follow the naming conventions remain in this category. For example, a variable name *cs* means nothing and does not provide any context for understanding. Therefore, this category does not concern the MMRUC3 approach, as the approach will not work effectively in this nonstandard projects. However, fixing the names of code entities is another research domain, and it is not within the scope of this article. We believe that the effectiveness of MMRUC3 will be increased if techniques such as those in the works of Høst and Østvold[43] and Allamanis et al[44] are used in the source code before applying MMRUC3. This observation has been left as a future work.

In order to categorize the standard of the projects (data sets), it is necessary to analyze the source code of each project. For this, two students of MSSE (Master of Science in Software Engineering) at the Institute of Information Technology, University of Dhaka, Bangladesh, are selected. They were not familiar with the seven source projects (shown in Table 1) and worked independently to prioritize and categorize the projects based on the naming conventions (or standards). They manually analyzed the projects by choosing several classes randomly from each package of a project (approximately 30% of the total classes for each project on an average). Thus, two category sets are generated independently by the two students. Finally, by merging the two sets, they generated the only one set of category for the project standards. Whenever

TABLE 10 Categorization of project standards

Project	Standard Category	Standard Value	Justification
1. JHotDraw	Excellent	1	Almost all classes follow the exact naming convention that is easily understandable and self-descriptive.
2. ArgoUML	Exellent	1	Almost all classes follow the exact naming convention that is easily understandable and self-descriptive.
3. JMeter	Best	2	Most of the classes contain completed and appropriate naming but a few names are not self-descriptive.
4. FreeMind	Better	3	Many classes follow appropriate name conventions. However, a few names should be completed to be easily understandable, such as *getMemLoad* should be *getMemoryLoad*.
5. Maven	Better	3	Many of the classes and methods follow naming conventions. However, some are nonstandard and hard to understand by name, for example, the *add2*() and *mergePluginContainer_Plugins*() methods.
6. DrJava	Good	4	Method naming conventions are followed in most cases, but variable naming conventions are not maintained properly, for example, *_active _scroll, _interpreterResetFailed*, etc.
7. Weka	Good	4	Naming convention is followed properly. However, some names are confusing to understand, such as *m_linearNormNorm*, *m_linearNormOrig*, etc.

TABLE 11 Categories of the source projects

Category	Project Id	Project
1. Excellent	{1, 2}	{JHotDraw, ArgoUML}
2. Best	{3}	{JMeter}
3. Better	{4, 5}	{FreeMind, Maven}
4. Good	{6, 7}	{DrJava, Weka}

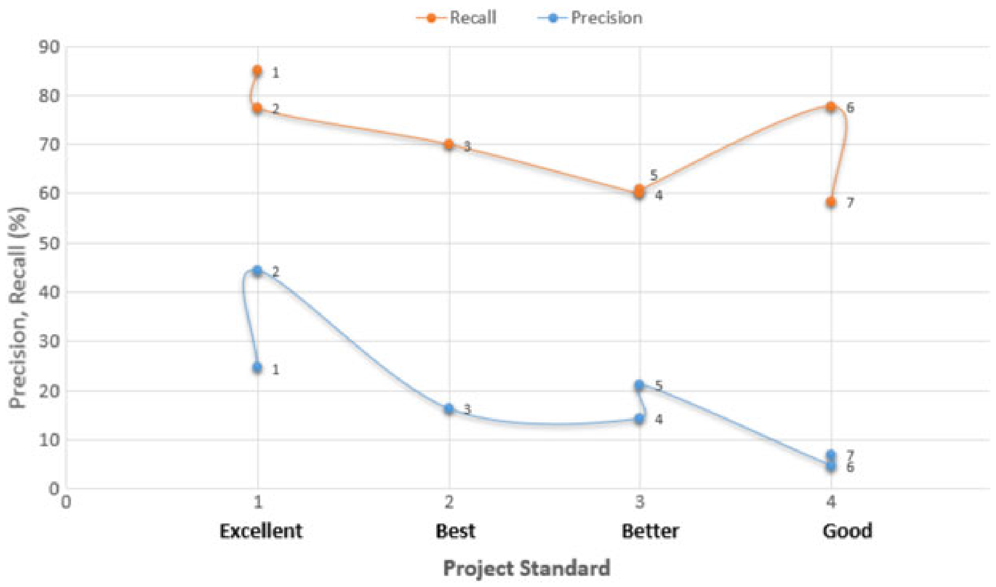


FIGURE 8 Coding standards versus precisions and recalls of MMRUC3 (the numbers shown in the graphs represent project id(s)) [Colour figure can be viewed at wileyonlinelibrary.com]

a conflicting issue arises, they consulted with each other and provided the final list. Their analysis for categorization is shown in Table 10, and the category list is shown in Table 11.

After the standards for each project are defined, patterns between *project standards* and the two accuracy metrics, namely, precisions and recalls, of MMRUC3 are shown graphically in Figure 8. The graph provides a significant finding as follows:

"*Since **project standards** decrease, both the two metric (precision and recall) values decrease.*"

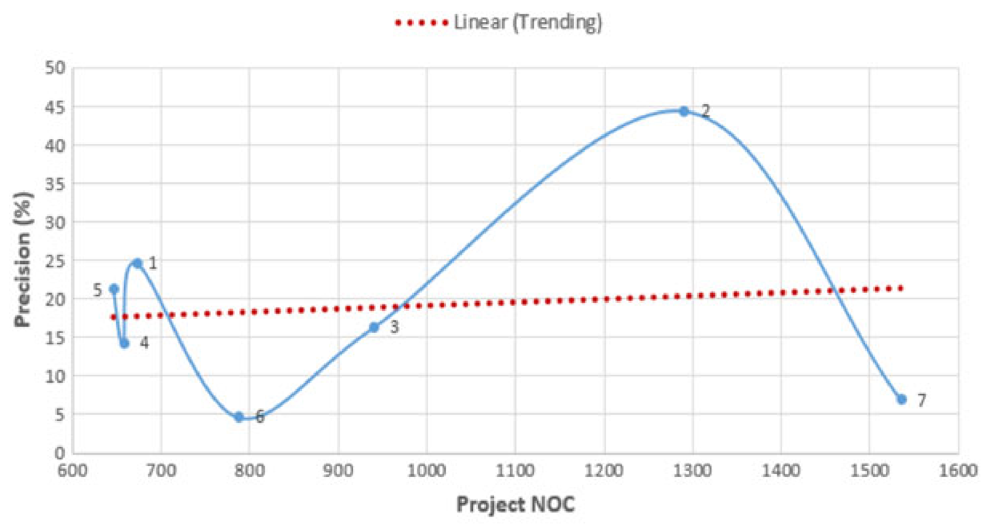
Therefore, the results are dependent on how well the project is written or developed, because the MMRUC3 approach considers the contextual information of the project for the *move method* refactoring recommendations.

5.3.2 | Relationship based on project sizes

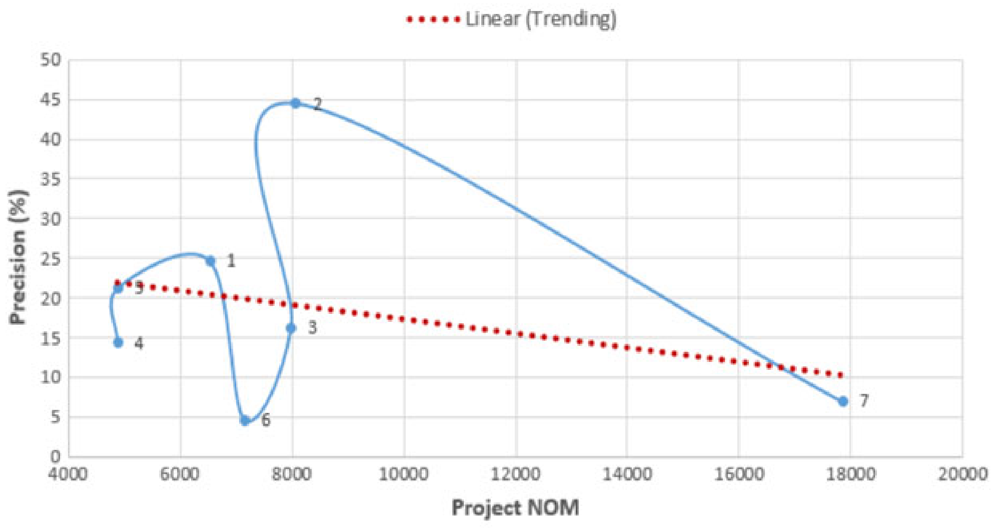
This subsection establishes the relationships between the proposed MMRUC3 approach and project sizes. More specifically, there exist a few patterns of precisions and recalls with project sizes: NOCs, NOMs, and LOCs. These patterns are shown graphically in Figures 9 and 10.

Figure 9 shows the relationships between the precisions of MMRUC3 and project sizes (NOCs, NOMs, and LOCs) as graphs. The graphs hardly find any pattern between the sizes of projects and the precisions of the approach, because the approach is dependent on project standards (ie, how well a project is written) rather than on the sizes. In the normal sense, it can be thought that there is a possibility of increasing false positive results as the project sizes increase, and hence, precisions could be lower. However, it can be seen in the graphs that the size metrics have no impact on the proposed MMRUC3 approach. The technique seems a balanced approach with respect to the experimental projects, as shown in the graphs. The finding of the graphs can be stated as follows:

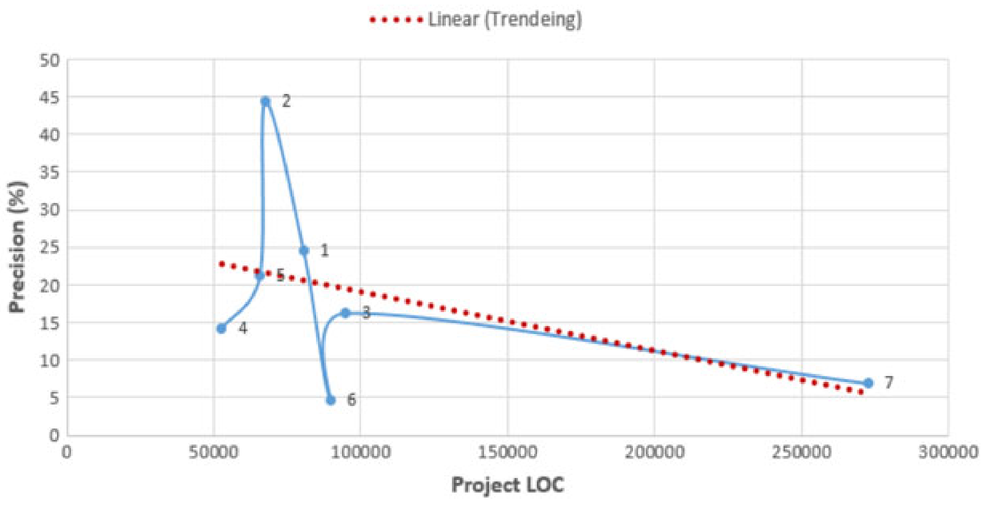
"There is no relationship between precisions of MMRUC3 and sizes (NOCs, NOMs, and LOCs) of projects, and so the approach is balanced."



(A)

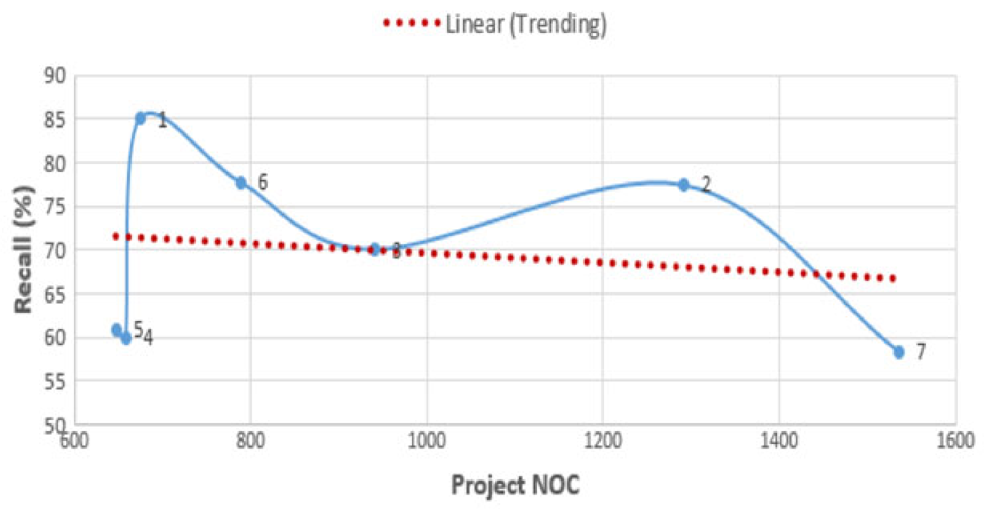


(B)

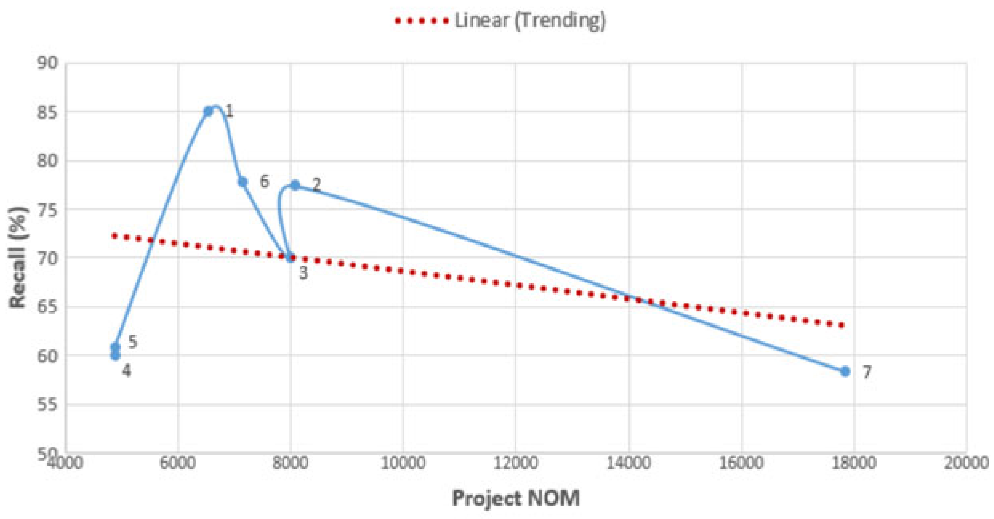


(C)

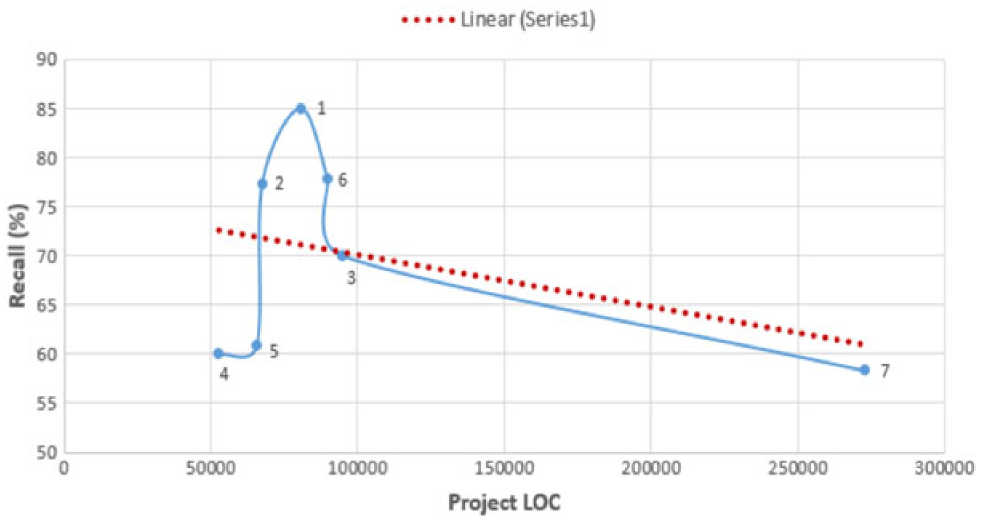
FIGURE 9 Patterns of precisions of MMRUC3 as sizes increase (the numbers shown in the graphs represent project id(s)). A, Number of classes (NOCs) versus precisions; B, Number of methods (NOMs) versus precisions; C, Line of codes (LOCs) versus precisions [Colour figure can be viewed at wileyonlinelibrary.com]



(A)



(B)



(C)

FIGURE 10 Patterns of recalls of MMRUC3 as sizes increase (the numbers shown in the graphs represent project id(s)). A, Number of classes (NOCs) versus recalls; B, Number of methods (NOMs) versus recalls; C, Line of codes (LOCs) versus recalls [Colour figure can be viewed at wileyonlinelibrary.com]

The graphs in Figure 9B,C indicate that it is hardly found any pattern of precisions, though the trends are downwards, when NOMs and LOCs increase. If it is analyzed more critically for the projects (project id 2, 3, and 6) and the projects (project id 2, 1, 3, and 6) having almost the same sizes of NOMs and LOCs, respectively, the precision values decrease on the basis of the project standards. These relationships are another significant finding of this research. Hence, the finding can be stated as follows:

"If NOM (number of methods) and LOC (line of codes) values are almost in the same range for a set of projects, then precision decreases on the basis of project standard (naming convention). That is, precision decreases with the deterioration of naming standard."

Similarly, the graphs in Figure 10 indicate that the trends of recalls decrease as values of NOC, NOM, and LOC increase, though it is hardly found any patterns of recalls of the projects. The main finding of these graphs is as follows:

"There is no direct relationship between recalls of MMRUC3 and sizes (NOCs, NOMs, and LOCs) of projects, and so the approach is balanced. If the values of the sizes are almost in the same range for a set of projects, then recall decreases on the basis of project standard (naming convention). That is, for those projects, recall decreases as the standard deteriorates."

5.4 | Comparative result analysis

The comparative results between the proposed approach and the widely used refactoring tool, ie, JDeodorant (an eclipse plugin), have been shown in this section. JDeodorant uses structural information (ie, coupling and cohesion) to detect and recommend *move method* refactoring. Thus, a comparison with the tool assists in perceiving the effectiveness of contextual information for recommending *move method* refactoring. Table 12 depicts the precision, recall, and F-measure obtained from the experimental data sets. The comparative analysis in terms of precision, recall, and F-measure is discussed below.

Result analysis in terms of precision: Figure 11 and Table 12 depict the precision values for all the seven experimental projects. It has been seen from the table that JDeodrant, which considers only coupling and cohesion, shows 15.93% precision on an average. However, MMRUC3 shows 18.91% precision on an average for the seven open source

TABLE 12 Comparative result analysis

		Precision, %		Recall, %		F-Measure, %	
Id No.	**Project**	**MMRUC3**	**JDeodorant**	**MMRUC3**	**JDeodorant**	**MMRUC3**	**JDeodorant**
1	JHotDraw	24.64	21.05	85.00	51.00	38.20	29.80
2	ArgoUML	44.44	37.5	77.42	56.25	56.47	45.00
3	JMeter	16.28	12.82	70.00	60.00	26.42	21.13
4	FreeMind	14.29	10.45	60.00	58.33	23.08	17.72
5	Maven	21.21	18.84	60.87	45.83	31.46	26.70
6	DrJava	4.64	4.25	77.78	72.22	8.75	8.03
7	Weka	6.90	6.57	58.33	64.52	12.33	11.93
	Average	***18.91***	***15.93***	***69.91***	***58.31***	***29.77***	***25.02***

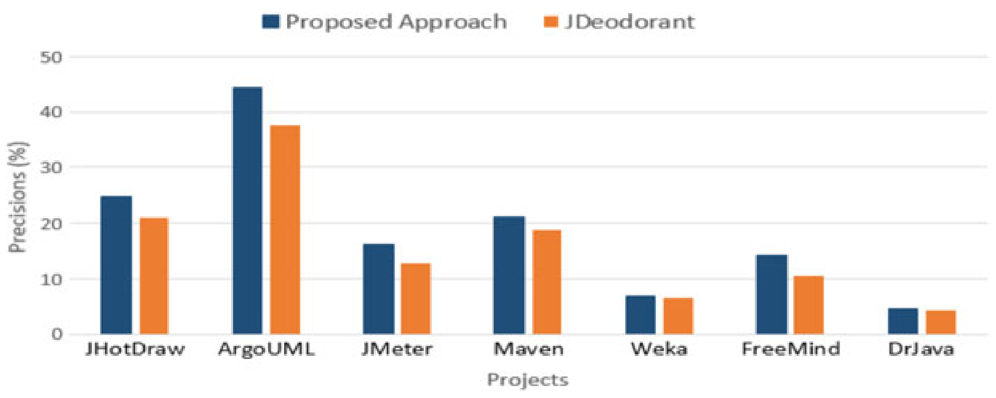


FIGURE 11 Comparison of precisions [Colour figure can be viewed at wileyonlinelibrary.com]

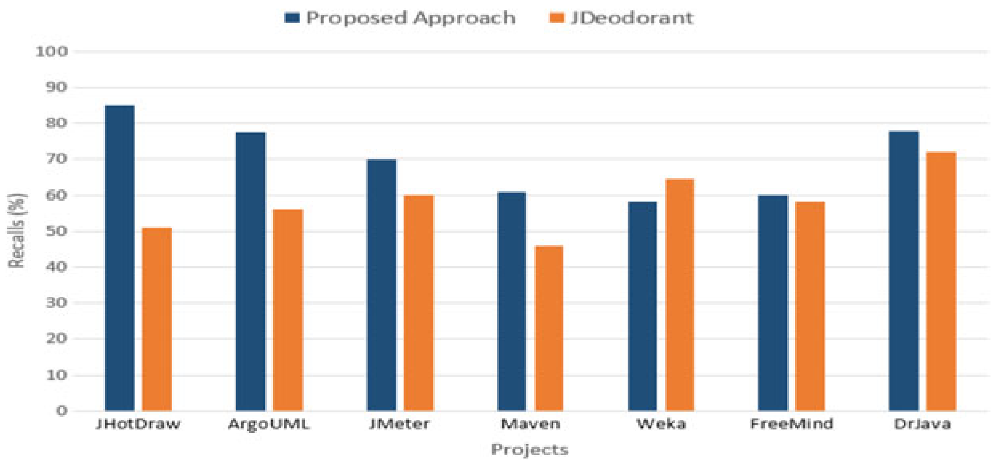


FIGURE 12 Comparison of recalls [Colour figure can be viewed at wileyonlinelibrary.com]

projects. It has also been seen in Figure 11 that MMRUC3 shows improvement in terms of precision for all the projects. It happens due to considering contextual information with structural metrics.

Result analysis in terms of recall: Recall values obtained for the experimental data sets have been shown in Table 12. A graphical representation of the obtained recall values has been shown in Figure 12 to perceive the comparison between JDeodorant and MMRUC3. According to the table, MMRUC3 produces 11.6% more recall on an average than JDeodorant. It clearly indicates the power of adding contextual information with the structural metrics (ie, coupling and cohesion). Although, MMRUC3 shows a higher recall value than JDeodorant for all the experimental projects except *Weka*. The reason is that *Weka* contains a lot of interactions, and the names of the code entities are not so much comprehensible like other projects. MMRUC3 depends on textual information of the code entities. Due to the lack proper naming convention, it is difficult for the approach to differentiate the contexts for all the methods of the project. However, it is expected that developers should follow proper naming conventions and MMRUC3 outperforms when conventions are properly followed as supported by the result analysis.

Balanced F-score: F-measure is a single measure that trades off precision versus recall. It is the weighted harmonic mean of precision and recall. If precision and recall are equally weighted, the balanced F-score (F_1*-score*) is found as the default F-measure. The F_1*-score* or F-measure (Equation 8) is calculated as the accuracy of the MMRUC3 recommendation approach.

$$F_1\text{-}score = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{8}$$

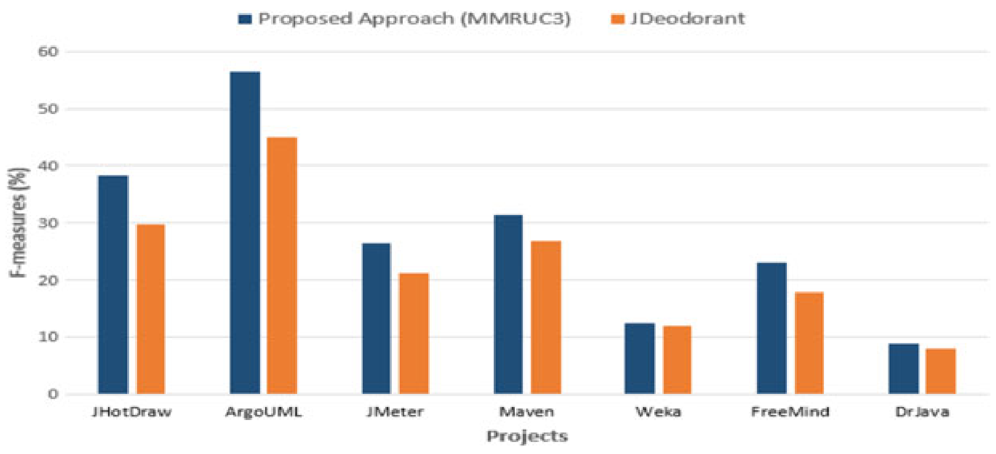


FIGURE 13 Comparison of F-measures [Colour figure can be viewed at wileyonlinelibrary.com]

As for example, the F-measure is calculated using average precision and recall for both of the techniques using Equation 8 as follows:

$$F_{\text{MMRUC3}} = 2 * \frac{18.91 * 69.91}{18.91 + 69.91} = 29.77\%$$

$$F_{\text{JDeodorant}} = 2 * \frac{15.93 * 58.31}{15.93 + 58.31} = 25.02\%.$$

Table 12 and Figure 13 depict the calculated F-measure for JDeodorant and MMRUC3. It has been seen that MMRUC3 shows improvement in terms of the F-measure for all the projects than JDeodorant. More precisely, it shows an approximately 4.75% more F-measure value on an average due to adding contextual information with structural metrics.

The experimental results show the evidence that, the inclusion of the contextual factor along with the dependencies (coupling and cohesion) in the recommendation process, is one of the most significant contributions of the research. Moreover, while the existing techniques have considered only non-static entities (methods and attributes), consideration of both static and non-static entities has made a meaningful improvement of the approach.

6 | CONCLUSION

Coupling and cohesion are the two key factors considered during the designing phase of a software application. Since developers only need to focus on coupled classes to meet the change requirements, the application should be loosely coupled and highly cohesive to make the maintenance task easier with lower effort, cost, and time. The feature envy code smell is a barrier to achieving this goal of the maintenance task, as it increases coupling and decreases cohesion. The proposed MMRUC3 approach plays a significant role in refactoring the code smell by recommending *move method* refactorings automatically. As a result, it helps optimize the design quality of the application in terms of coupling and cohesion. Moreover, it assists software engineers in their maintenance activities in terms of lesser effort, time, and cost. The MMRUC3 approach, relying on both dependency- and contextual-based information for both static and nonstatic entities (method and attributes), improves the recommendation accuracy with better precision, recall, and F-measure than the competitive refactoring tool. In addition, the approach helps increase software modularization and follow the SRP through optimizing interactions among the components of the application. It is stated that if the project architecture is standard and component based, that is, it contains small classes based on the responsibilities, then our approach also persists in this architecture by suggesting methods to the appropriate classes based on the responsibilities. Thus, similar methods having similar tasks should be in the same class. Moreover, methods are distributed according to their responsibilities. Therefore, low coupling and high cohesion are achieved if the suggested refactorings have been performed. The incorporation of *C3* factors, namely, coupling, cohesion, and contextual information, in this article makes the approach different from the existing traditional approaches. The *C3* factors and relationships between project nature (sizes and standards) and the proposed approach assist the researchers and software engineering practitioners in developing more new refactoring tools, approaches, and ideas.

6.1 | Threats to validity

The internal, external, and conclusion threats to validity of this study are discussed in this section.

Threats to internal validity: Subjectiveness in the categorization of project standards is inevitable due to a lot of manual efforts involved in the experimental study. In addition, there also might be human errors in analyzing coding standards (naming conventions) of the source projects. These threats have been mitigated by independently double-checking all manual work. It is ensured that the results are individually verified and agreed upon by two MSSE students. These threats could be further reduced by involving third-party people who have experiences on coding standards to verify the results.

Threats to external validity: An external validity threat is that we cannot extrapolate our results to other projects. In addition, the relationships between the accuracy of MMRUC3 and coding standard and project sizes are established on the basis of seven open source projects. The application of more projects could minimize this threat, but lacking are projects related to the refactorings.

Threats to conclusion validity: We compare our results (precisions and recalls) with the results provided in the online repositories (seven projects) for the validation of the approach. Those online results could miss the identification of some smelly instances. As these are mostly used repositories for the refactoring domain, we claim that the number of such instances is really small. In addition, there could be a situation where the dependency-based similarity might suggest a class and the context-based similarity might suggest another one. Hence, the outcome of the combined similarities could provide an inconsistent suggestion for refactoring in this kind of situation. However, if the project is well developed following appropriate naming conventions, then the two similarity calculations might go in the same direction separately, ie, suggest the same class, and hence, the combined calculation does so. Moreover, the two similarity processes help each other when two or more classes get almost the same similarity score by only one similarity measurement process.

6.2 | Future direction

The proposed solution and algorithm can be extended in a few interesting directions.

- In this article, both the dependency (coupling and cohesion)-based and context-based similarity factors have the same priority in the similarity measurement process of the approach. The future plan is to set priority on the two similarity factors in order to analyze the effectiveness of the two factors separately.
- As the contextual factor of an application provides useful information, the plan is to utilize this factor in other refactoring techniques, such as *extract method, extract class,* etc, which are dependent on contexts of components of an application.
- Investigating the behavior of MMRUC3 by preprocessing the source code through fixing the irrelevant names of code entities using the techniques in the works of Høst and Østvold[43] and Allamanis et al.[44]

ACKNOWLEDGEMENT

This research was supported by the ICT Division, Government of the People′s Republic of Bangladesh, Bangladesh under Grant 56.00.0000.028.33.065.16-747, 21-06-2016 and Grant 56.00.0000.028.33.065.16-747, 14-06-2016.

ORCID

Md. Masudur Rahman http://orcid.org/0000-0002-0931-1919

REFERENCES

1. Martin F, Beck K, Brant J. *Refactoring: Improving the Design of Existing Code*. Boston, MA: Addison-Wesley Professional; 1999.
2. Sjøberg D, Yamashita A, Anda BCD, Mockus A, Dybå T. Quantifying the effect of code smells on maintenance effort. *IEEE Trans Softw Eng*. 2013;39(8):1144-1156.
3. Moser R, Abrahamsson P, Pedrycz W, Sillitti A, Succi G. A case study on the impact of refactoring on quality and productivity in an agile team. *Balancing Agility and Formalism in Software Engineering: Second IFIP TC 2 Central and East European Conference on Software Engineering Techniques, CEE-SET 2007, Poznan, Poland, October 10-12, 2007, Revised Selected Papers*. Berlin, Germany: Springer-Verlag Berlin Heidelberg; 2008:252-266.
4. Yamashita A, Moonen L. To what extent can maintenance problems be predicted by code smell detection?–an empirical study. *Inf Softw Technol*. 2013;55(12):2223-2242.
5. Yamashita A, Moonen L. Exploring the impact of inter-smell relations on software maintainability: An empirical study. Paper presented at: 2013 35th International Conference on Software Engineering (ICSE); 2013; San Francisco, CA.
6. Fokaefs M, Tsantalis N, Chatzigeorgiou A. Jdeodorant: Identification and removal of feature envy bad smells. Paper presented at: 2007 IEEE International Conference on Software Maintenance; 2007; Paris, France.
7. Sales V, Terra R, Miranda LF, Valente MT. Recommending move method refactorings using dependency sets. Paper presented at: 2013 20th Working Conference on Reverse Engineering (WCRE); 2013; Koblenz, Germany.
8. Liu H, Wu Y, Liu W, Liu Q, Li C. Domino effect: Move more methods once a method is moved. Paper presented at: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER); 2016; Suita, Japan.
9. Simon F, Steinbruckner F, Lewerentz C. Metrics based refactoring. In: Proceedings of the Fifth European Conference on Software Maintenance and Reengineering; 2001; Lisbon, Portugal.
10. Martin RC. *Agile Software Development: Principles, Patterns, and Practices*: Upper Saddle River, NJ: Prentice Hall PTR; 2003.
11. Melville P, Sindhwani V. Recommender systems. *Encyclopedia of Machine Learning*. Boston, MA: Springer Science+Business Media, LLC; 2011:829-838.
12. Manning CD, Raghavan P, Schütze H. *Introduction to Information Retrieval*. Vol. 1. Cambridge, UK: Cambridge University Press; 2008.
13. Jmovw. Applied Software Engineering Research Group. http://aserg.labsoft.dcc.ufmg.br/jmove/. Accessed March 20, 2017.
14. Tsantalis N, Chatzigeorgiou A. Identification of move method refactoring opportunities. *IEEE Trans Softw Eng*. 2009;35(3):347-367.

15. Oliveto R, Gethers M, Bavota G, Poshyvanyk D, De Lucia Andrea. Identifying method friendships to remove the feature envy bad smell (NIER track). In: Proceedings of the 33rd International Conference on Software Engineering; 2011; Honolulu, HI.
16. Palomba F, Bavota G, Di Penta M, Oliveto R, De Lucia A, Poshyvanyk D. Detecting bad smells in source code using change history information. In: Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering; 2013; Silicon Valley, CA.
17. Palomba F, Bavota G, Di Penta M, Oliveto R, Poshyvanyk D, De Lucia A. Mining version histories for detecting code smells. *IEEE Trans Softw Eng*. 2015;41(5):462-489.
18. Marinescu R, Ganea G, Verebi I. inCode: Continuous quality assessment and improvement. Paper presented at: 2010 14th European Conference on Software Maintenance and Reengineering; 2010; Madrid, Spain.
19. Hamid A, Ilyas M, Hummayun M, Nawaz A. A comparative study on code smell detection tools. *Int J Adv Sci Technol*. 2013;60:25-32.
20. Kimura S, Higo Y, Igaki H, Kusumoto S. Move code refactoring with dynamic analysis. Paper presented at: 2012 28th IEEE International Conference on Software Maintenance (ICSM); 2012; Trento, Italy.
21. Napoli C, Pappalardo G, Tramontana E. Using modularity metrics to assist move method refactoring of large systems. Paper presented at: 2013 Seventh International Conference on Complex, Intelligent, and Software Intensive Systems; 2013; Taichung, Taiwan.
22. Carneiro GF, Silva M, Mara L, et al. Identifying code smells with multiple concern views. Paper presented at: 2010 Brazilian Symposium on Software Engineering; 2010; Salvador, Brazil.
23. van Emden E, Moonen L. Java quality assurance by detecting code smells. Paper presented at: Ninth Working Conference on Reverse Engineering, 2002. Proceedings; 2002; Richmond, VA.
24. Fontana FA, Mäntylä MV, Zanoni M, Marino A. Comparing and experimenting machine learning techniques for code smell detection. *Empir Softw Eng*. 2016;21(3):1143-1191.
25. Deerwester S, Dumais ST, Furnas GW, Landauer TK, Harshman R. Indexing by latent semantic analysis. *J Am Soc Inf Sci*. 1990;41(6):391.
26. Baeza-Yates R, Ribeiro-Neto B. *Modern Information Retrieval*. Vol. 463. New York, NY: ACM Press; 1999.
27. Bavota G, De Lucia A, Marcus A, Oliveto R. A two-step technique for extract class refactoring. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering; 2010; Antwerp, Belgium.
28. Bavota G, Oliveto R, De Lucia A, Antoniol G, Guéhéneuc Y-G. Playing with refactoring: Identifying extract class opportunities through game theory. Paper presented at: 2010 IEEE International Conference on Software Maintenance; 2010; Timisoara, Romania.
29. Bavota G, Gethers M, Oliveto R, Poshyvanyk D, De Lucia A. Improving software modularization via automated analysis of latent topics and dependencies. *ACM Trans Softw Eng Methodol (TOSEM)*. 2014;23(1):4.
30. Ponisio L, Nierstrasz O. *Using Contextual Information to Assess Package Cohesion* [Technical Report]. Bern, Switzerland: Institute of Applied Mathematics and Computer Sciences, University of Berne; 2006.
31. Gethers M, Poshyvanyk D. Using relational topic models to capture coupling among classes in object-oriented software systems. Paper presented at: 2010 IEEE International Conference on Software Maintenance; 2010; Timisoara, Romania.
32. Poshyvanyk D, Marcus A, Ferenc R, Gyimóthy T. Using information retrieval based coupling measures for impact analysis. *Empir Softw Eng*. 2009;14(1):5-32.
33. Horstmann CS, Cornell G. *Core Java 2: Volume I, Fundamentals*. London, UK: Pearson Education; 2002.
34. Niwattanakul S, Singthongchai J, Naenudorn E, Wanapu S. Using of Jaccard coefficient for keywords similarity. In: Proceedings of the International Multiconference of Engineers and Computer Scientists; 2013; Hong Kong.
35. Papadimitriou CH. Computational complexity. *Encyclopedia of Computer Science*. Chichester, UK: John Wiley and Sons Ltd; 2003.
36. Gamma E. *Design Patterns: Elements of Reusable Object-Oriented Software*. Chennai, India: Pearson Education; 1995.
37. ByteParser. https://github.com/rifatbit0401/ByteParser. Accessed November 10, 2016.
38. JavaParser - Home. http://javaparser.org/. Accessed January 12, 2017.
39. Woodcock J, Davies J. *Using Z: Specification, Refinement, and Proof*. Vol. 39. Englewood Cliffs, NJ: Prentice Hall; 1996.
40. Satter A, Ami AS, Sakib K. A static code search technique to identify dead fields by analyzing usage of setup fields and field dependency in test code. Paper presented at: The 3rd International Workshop on Concept Discovery in Unstructured Data; 2016; Moscow, Russia.
41. Butler S, Wermelinger M, Yu Y, Sharp H. Relating identifier naming flaws and code quality: An empirical study. Paper presented at: 2009 16th Working Conference on Reverse Engineering; 2009; Lille, France.
42. Deißenbock F, Pizka M. Concise and consistent naming [software system identifier naming]. Paper presented at: 13th International Workshop on Program Comprehension (IWPC'05); 2005; St. Louis, MO.
43. Høst EW, Østvold BM. Debugging method names. Paper presented at: European Conference on Object-Oriented Programming; 2009; Genoa, Italy.
44. Allamanis M, Barr ET, Bird C, Sutton C. Suggesting accurate method and class names. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering; 2015; Bergamo, Italy.

How to cite this article: Rahman MM, Riyadh RR, Khaled SM, Satter A, Rahman MR. MMRUC3: A recommendation approach of *move method* refactoring using coupling, cohesion, and contextual similarity to enhance software design. *Softw Pract Exper*. 2018;1–28. https://doi.org/10.1002/spe.2591