International Journal of Software Engineering
and Knowledge Engineering
Vol. 30, No. 9 (2020) 1289–1307

DOI: 10.1142/S0218194020500357

World Scientific
www.worldscientific.com

ABMMRS Eradicator: Improving Accuracy in Recommending Move Methods for Web-based MVC Projects and Libraries Using Method's External Dependencies

Atish Kumar Dipongkor

Department of Computer Science and Engineering
Jashore University of Science and Technology
Jashore 7408, Bangladesh
atish.cse@just.edu.bd

Iftekhar Ahmed*, Rayhanul Islam†, Nadia Nahar‡,
Abdus Satter§ and Md. Saeed Siddik¶

Institute of Information Technology
University of Dhaka
Dhaka 1000, Bangladesh
**iftekhar.ahmed@rwth-aachen.de*
†*rayhanul.islam@du.ac.bd*
‡*nadia@iit.du.ac.bd*
§*abdus.satter@iit.du.ac.bd*
¶*saeed.siddik@iit.du.ac.bd*

Received 2 February 2020
Revised 21 May 2020
Accepted 22 June 2020

Move Method Refactoring (MMR) is used to place highly coupled methods in appropriate classes for making source code more cohesive. Like other refactoring techniques, it is mandatory that applying MMR will preserve applications' behaviors. However, traditional MMR techniques failed to meet this essential precondition for Action methods in web-based application and API methods in libraries projects. The reason is that applying MMR on these methods changes the behaviors of the projects by raising Application-breaking issues, for instance, failure of browser requests and compilation errors in client projects. To resolve this problem, developers are suggested to manually check Action and API methods while applying MMR. However, manually inspecting thousands of lines of code for these issues is a time-consuming and hectic task. In this paper, an advanced MMR technique is proposed which automatically identifies Application-breaking MMR suggestions. This technique first takes the initial move method suggestions from the existing prominent MMR techniques e.g. JDeodorant. For each of the suggestions, it parses the source code and construct Abstract Syntax Tree to examine two types of usage. One is whether a suggestion has not been used in any unit test and Regular Class, and another is whether the suggestion has been used in unit test classes only. If any MMR suggestion is found having one of these two types of usage or both, the respective suggestion is marked as

Application-breaking. In order to evaluate the proposed technique, several experiments have been conducted on open source projects. The experimental results show that the proposed technique achieved 96.4% Precision, 90% Recall and 93.1% *F*-score in detecting Application-breaking MMR suggestions, because of considering external dependencies of the MMR suggestions.

Keywords: Feature envy; move method; refactoring; Application-breaking.

1. Introduction

Code Smells are treated as design problem because these degrade the quality, understandability and changeability of code. Moreover, these require extra cost, time, and effort in software development and maintenance [1–3]. Fowler and Beck [4] identified 22 different code smells; among those Feature Envy (FE) is one of the most common and recurring ones [5, 6]. FE occurs in code when a method makes too many calls to the methods of another class to obtain data and/or functionality [4]. The main consequence of FE is that it makes software components highly coupled and loosely cohesive. Considering the impact of high coupling and low cohesion [7–11], FE should be identified and eliminated from source code through Move Method Refactoring (MMR) techniques. Nevertheless, due to Application-breaking issues, it is not possible to apply MMR on the FE instances like Action[a] methods of web-based Model-View-Controller (MVC) projects [12–14] and API[b] methods of libraries [15]. The Action methods of web-based MVC projects become coupled with other classes in order to serve the requests from web browsers and eventually, manifest themselves as FE instances. However, applying MMR on these methods breaks web-based MVC applications' structure and behavior such as failure of browser requests and compilation errors. Similarly, API methods of libraries [15] become coupled with other classes for serving client projects and finally, reveal themselves as FE instances. However, it is not possible to apply MMR on these instances because applying MMR breaks the functionalities of client projects such as compilation errors. According to Mens *et al.* [16], the automated refactoring techniques should guarantee that the applied refactoring will preserve applications' behavior, but the existing techniques of MMR do not work properly in the above situations. To this end, MMR should not be applied to the FE instances like Action and API methods for preserving the behavior of web-based MVC and libraries.

To increase the accuracy of Move Method recommendation, it is required to identify and eliminate Action and API methods from Move Method Refactoring Suggestion (MMRS). Although developers can identify and eliminate these suggestions manually while refactoring, it is a time-consuming and error-prone task. So, an automated technique is required to alleviate these problems. However, it is challenging to automatically detect Action and API methods because the syntactic information of Action and API methods varies across languages and frameworks. For

[a] Action methods have a one-to-one mapping with browser requests.
[b] These methods execute appropriate application logic.

instance, Action methods in Java Spring Framework [17], must have a predefined annotation but that is not the case for Action methods in ASP.NET Framework [18]. In addition, some languages (e.g. C#, Java and PHP) do not have syntactic information for API methods at all.

Several techniques have been proposed in the literature for recommending MMRS. One of the most commonly used techniques named JDeodorant was proposed by Tsantalis *et al.* to identify FE instances based on their entity usages from other classes [19]. Simon *et al.* proposed a technique using distance-based cohesion metric for identifying MMR opportunities in small applications [20]. Jehad Al Dallal described an approach to predict MMR opportunities in object-oriented software systems [21]. Other notable techniques such as JMove [22], TACO [23], HIST [24] and MethodBook [25] have been found in the literature for Move Method Recommendations. All the techniques provide promising MMRSs but their suggestions are not applicable in web-based MVC projects and libraries. The reason is that applying MMR based on these techniques ends up breaking the behavior of libraries and web-based MVC projects.

In this paper, a technique is proposed for identifying Application-breaking Move Method Refactoring Suggestion (ABMMRS) in web-based MVC projects and libraries. At first, the technique takes source code of the input project and parses Regular and Unit test classes from the code using Abstract Syntax Tree (AST). The Regular classes implement core functions of an application using entities (methods or attributes) of their own or from other classes. On the other hand, unit test classes evaluate the functions of Regular classes. Next, it obtains FE instances using an existing technique, for example, JDeodorant [19]. For each of the instances, the proposed technique verifies whether it has been used by any of the Regular and Unit test classes. If it is not used by any of the Regular and Unit test classes, or only used by unit test classes, it is marked as an ABMMRS. In this study, the assumption is that an FE instance must be externally connected when it is not in use in internal code (or only in test code). Therefore, any refactoring of FE instances will break their external connections, which in turn will break the application. Due to this Application-breaking issue, the proposed technique eliminates all ABMMRSs from already obtained FE instances and suggests the rest of FE instances for MMR.

In order to evaluate the proposed technique, a tool named ABMMRS Eradicator is developed, and an empirical study is conducted on nine open-source software projects (five web-based MVC projects and four libraries). ABMMRS Eradicator uses a well-known existing technique named JDeodorant [19] in order to obtain initial MMRSs. Our study result shows that JDeodorant provided in a total of 91 MMRSs among which 30 were ABMMRS. Twenty eight out of these 30 ABMMRS were identified by ABMMRS Eradicator, which is almost 93.33%. The overall effectiveness of ABMMRS Eradicator in identifying and eliminating ABMMRS consists of 96.4% Precision, 90% Recall and 93.1% *F*-score.

The rest of this paper is organized as follows. Section 1.1 discusses a motivational example from the real world. Section 2 describes some notable works which are highly

related to this research. In Sec. 3, we proposed a technique named ABMMRS Eradicator is demonstrated. Section 4 discusses the evaluation and findings of the proposed technique. Some validity threats are shown in Sec. 5. Finally, this paper is concluded in Sec. 6 with future directions.

1.1. *Motivation: Broken application after applying MMR*

Web-based MVC frameworks consist of three major components, namely Model, View and Controller [12–14]. In these frameworks, View takes user input from web browser and send them to the Action method of a Controller. Then, the Action method communicates with Model for processing the input. Another important note about Web-based MVC frameworks is that a View always has one-to-one mapping with a specific Action method. In Fig. 1, *sendInvites*() is an Action method of a Controller named *MailInviteController*. It serves the browser's requests by calling two methods, namely *getInvites*() and *getInvitationText*() from a Model named *MailInviteForm*. As *sendInvites*() is coupled with *MailInviteForm*, existing approaches [19, 22, 25] identify it as an FE instance and suggest that MMR should be applied on *sendInvites*() to decouple it from the envied class. However, if *getInvites*() is moved to *MailInviteForm* based on the suggestion of existing techniques, it ends up with the following issues:

- Application is broken because browser tries to map the request to *MailInvite Controller* but *sendInvites*() is moved to *MailInviteForm*. In MVC framework, Model classes are unable to serve browser request.
- Moving Action methods to Model is a violation of MVC architecture [26].

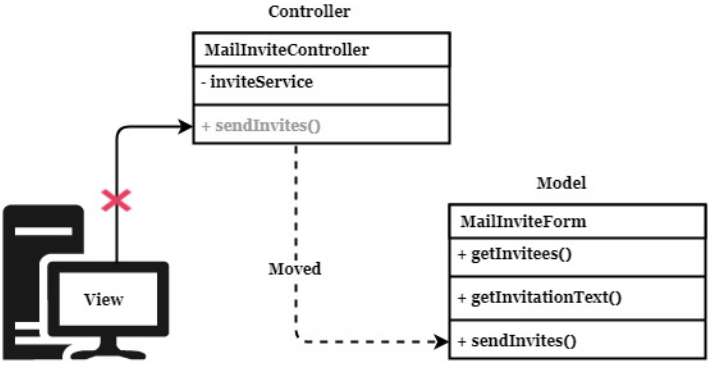

Fig. 1. The broken application after applying MMR on an Action method.

To avoid the above issue, another approach can be used to decouple FE instances. For example, *getInvites*() and *getInvitationText*() can be moved to *MailInviteController*. As a result, application will not break as well as the FE

instance *sendInvites*() will no longer be coupled with *MailInviteForm*. However, moving all envied methods (*getInvites*() and *getInvitationText*()) to source class (*MailInviteController*) may yield another type of Code Smell in source class, namely God Class [27]. In addition, it is needed to copy the same methods *getInvites*() and *getInvitationText*() to *InviteController* as one of its Action methods *acceptInvite*() is coupled with them. If it is done, coupling between *acceptInvite*() and *MailInvite Form* will be greatly reduced. However, writing redundant code (*getInvites*() and *getInvitationText*() are already implemented in *MailInviteController*) violates a well-known software principle named Do not Repeat Yourself (DRY) [28].

The same inevitable issues are experienced while applying MMR on the API methods of libraries. In Fig. 2, *getObject*() is an API method located in *JestClient Factory* class of Jest[c] library. This method is an FE instance because it calls six methods from *DroidClientConfig*. Although *getObject*() is tightly coupled with *DroidClientConfig*, it is not possible to follow the MMRSs of existing techniques . [19, 22, 25] The reason is that if *getObject*() is moved to *DroidClientConfig*, all the client projects that are using *getObject*() from *JestClientFactory* will break. To resolve coupling in *getObject*(), we can move all 16 envied methods from *Droid ClientConfig* to *JestClientFactory*. However, this resolution will cause God Class as in the first example.

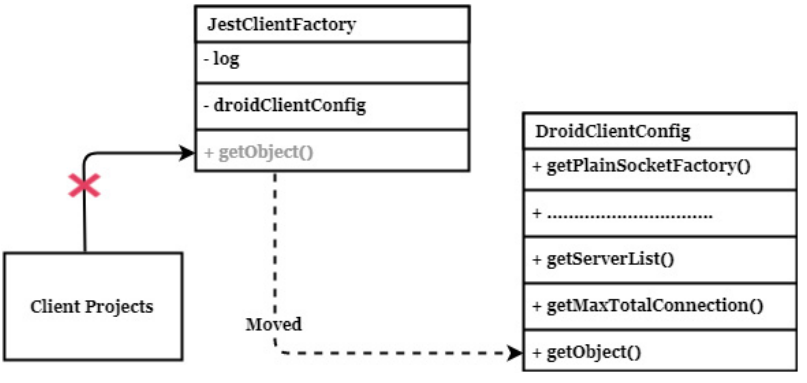

Fig. 2. The broken application after applying MMR on an API method.

Although Action and API methods are FE instances according to the nature of their applications, it is not possible to apply MMR because either it breaks applications or produces God Class in code. It is an oath for refactoring any application that refactoring must improve code quality without breaking the application's behavior. However, there was no existing technique found which works seamlessly in the above circumstances.

[c]Jest is a popular HTTP REST client for ElasticSearch.

2. Literature Review

MMR is drawing the attention of researchers for many years due to coupling and cohesion issues. In this section, some notable techniques will be discussed with their strengths and weaknesses. Besides, these techniques are categorized based on their methodologies such as (a) MMR using coupling and cohesion and (b) MMR based on similarity.

2.1. *MMR using coupling and cohesion*

The techniques discussed in this section identify FE instances based on the degree of coupling.[d] Besides, they provide MMRSs to reduce coupling and increase cohesion among software components. However, some major limitations are observed which expedite opportunities for further work.

Simon *et al.* [20] proposed a distance-based cohesion metric to identify Move Method and attribute situations. According to this approach, an entity (method or attribute) should be moved if it uses or it gets used by other entities of another class than its own. However, this approach identifies Move Method and attribute situations without examining any behavior-preserving preconditions [29] As this approach does not analyze behavior-preserving preconditions, the consideration of its suggestions will break the applications' behavior.

JDeodorant [19] is an automated tool which identifies FE instances and provides MMRSs based on the usage of entities. For instance, a method m is an FE instance and it should be moved to another C_i if it calls/uses more entities from C_i than its current class C. Moreover, this is the very first technique that examines behavior-preserving preconditions (Compilation, Behavior-Preservation and Quality) before providing MMRSs for the identified FE instances. However, JDeodorant provides MMRSs on Action and API methods which break MVC projects and the client program of libraries after refactoring. The reason is that Action and API methods have external dependencies which are not considered in the preconditions of JDeodorant. Thus, it becomes a manual task for developers to inspect the MMRSs of JDeodorant before refactoring.

Jehad Al Dallal [21] applied a statistical technique to predict MMR opportunities in Object-Oriented Code. This technique considered both coupling and cohesion aspects of methods while identifying MMR scopes. Although the author of this technique claimed high precision of his technique, he did not provide any comparative study with the existing techniques. In addition, to preserve the applications' behavior this technique borrowed all preconditions from JDeodorant which are not sufficient for MVC and library projects. Thus, consideration of this technique for refactoring will break the behavior of MVC and library projects.

[d]How many methods or attributes are used from other classes.

2.2. *MMR based on similarity*

All the approaches reviewed in this section provide MMRSs based on some similarity measurements, i.e. structural, semantical and so on. According to these approaches, a method should be moved if it is more similar (in terms of structural and semantical dependencies) to the methods of another class than its source class. To the best of our knowledge, the issues mentioned in Sec. 2.1 also exist in these techniques.

JMove [22] provides MMRSs to remove FE instances by calculating similarity coefficient [30, 31] of dependency sets. Firstly, for a given method m, it calculates similarity coefficient with all methods of its current class. Then, it calculates similarity coefficient with rest of the methods in other classes. If the similarity coefficient is higher in a class other than its current class and m satisfies all behavior-preserving preconditions proposed by JDeodorant [19], this approach provides MMRS for m. It is worth noting that JMove is totally dependent on JDeodorant's preconditions to preserve applications' behavior. As JDeodorant's preconditions cannot guarantee the behavior of MVC and client applications, consideration of this approach for MMR will also break applications (MVC and libraries). Other limitations of this approach are (a) its MMRSs contains a high percentage of false-positive compared to JDeodorant and (b) it cannot provide appropriate suggestion when the similarity coefficient becomes equal in multiple classes.

MMRUC3 [32] is another JMove alike technique of providing MMRSs for eradicating FE instances from source code. With compared to JMove, the main difference is that this technique uses a different similarity measurement, i.e. Jaccard similarity and besides, it considers contextual similarity along with the dependency sets similarity. Although this approach introduced two new things, e.g. Jaccard and Contextual similarity, it did not provide any justification for that. The main limitation of this approach is that it does not examine any precondition before providing MMRSs. Therefore, this approach will fail to preserve the applications' behavior after refactoring.

Methodbook [25] identifies MMR opportunities using Relational Topic Models (RTM) [33] for removing FE instances. According to this approach, a method m should be moved to another class C_i if C_i contains more friends of m than its own class C. To find friends of the method m, Methodbook considers three types of similarities among the methods of other classes; using same attributes, calling same methods, and semantic similarity.[e] The evaluation of Methodbook reported that it was not able to achieve better than JDeodorant. Moreover, it borrowed all the preconditions from JDeodorant to preserve applications' behavior. Hence, it has the same improvement scope (Application-breaking issue) like JDeodorant.

TACO [23] removes FE instances based on the textual similarity of methods. If a method m is textually similar to another class C_i than its current class C, then m should be moved to C_i. Textual similarity among methods is calculated using Cosine similarity. The main limitations of this approach are (a) it does not examine any

[e]Similarity in code documentation (e.g. javadoc) between two methods.

behavior-preserving preconditions before providing MMRSs which will certainly break applications' behavior, (b) it cannot provide accurate MMRSs without enough textual information and (c) it provides multiple MMRSs when similarity becomes equal with multiple classes.

HIST [24] provides MMRSs based on historical co-changes for eliminating FE instances from source code. According to this approach, if a method m changes more frequently when changes occur in its dependent class C_i then it should be moved to C_i. The main limitations of this approach are (a) there is no direction for how applications' behavior will be preserved, (b) it cannot provide accurate MMRSs without enough historical information and (c) it does not consider any structural information (i.e. whether m calls any methods and/or attributes from C_i or not).

3. Proposed Technique

In this paper, a technique named ABMMRS Eradicator has been proposed for identifying and eliminating ABMMRSs in MVC and library projects. As a result, developers will be able to apply non-ABMMRSs to their applications automatically. Initially, the ABMMRS are mathematically formalized using *Z* [34] notations. Then, an algorithm has been proposed to identify and eliminate ABMMRS automatically.

3.1. *Mathematical formalization of ABMMRS*

Having said that Action and API methods of MVC and libraries, respectively, are ABMMRS, it is required to analyze the structural and syntactic behaviors of these methods for mathematical formalization. Action methods in Java Spring Framework [17] always have an annotation [35] called *RequestMapping*, and it can be considered a way of identifying ABMMRS in MVC projects. However, Action methods in ASP. NET MVC Framework [18] do not hold any annotation. Instead, these methods are marked by their return type which must be an instance of the ActionResult [36] class. Moreover, the libraries' API methods neither have any annotation nor their return types derive from a specific class. At this point, it is found that semantic information of Action and API methods is not helpful for identifying ABMMRS as it varies across frameworks and platforms. Therefore, structural information of these methods is analyzed and the following common doings among all frameworks and platforms are found.

- These methods are always invoked by external entities, i.e. web browsers and client applications invoke Action and API methods, respectively.
- These methods are obedient to hold *public* access modifier as external entities cannot invoke a method if its access modifier is not *public*.
- These methods have no internal caller classes except the unit test classes. These methods are only called/used by the unit test classes for unit testing.

Based on the above common structural behaviors, the ABMMRS are formalized using *Z* [34] notations. Let us say, M is a set that contains all MMRSs of a given system S (MVC or library projects), and $m?$ a member of M. Moreover, M only contains the methods having *public* access modifier. Now, it is the objective of our proposed technique to identify whether $m?$ is an ABMMRS or not. To this end, we define a boolean free type [34] ABMMRS. The value of ABMMRS can be *true* or *false*.

$$\text{ABMMRS} ::= true | false. \tag{1}$$

The classes of S can be grouped into two classes, namely Regular classes[f] ($\mathbb{P}Class$) and unit test classes[g] ($\mathbb{P}TestClass$). The members of Regular and Unit test classes is denoted as $C_{\text{caller}} \in \mathbb{P}Class$ and $T_{\text{caller}} \in \mathbb{P}TestClass$, respectively. Now, $m?$ may go through the following situations:

(1) $m?$ can or cannot be invoked by a Regular class which is denoted as a relation, $R_C : m? \rightarrow C_{\text{caller}}$.
(2) $m?$ can or cannot be invoked by a unit tests class which is denoted as a relation, $R_T : m? \rightarrow T_{\text{caller}}$.
(3) $m?$ is not getting invoked by either any C_{caller} or T_{caller}. In that case, the ranges of R_C and R_T relations will be empty, i.e. $\text{range}(R_C) = \emptyset$ and $\text{range}(R_T) = \emptyset$. This scenario indicates that $m?$ has an external invoker as there is none from $\mathbb{P}Class$ and $\mathbb{P}TestClass$.
(4) $m?$ is only invoked by T_{caller} but not by C_{caller} which means the range of R_T is not empty but the range of R_C is empty, i.e. $\text{range}(R_T) \neq \emptyset$ and $\text{range}(R_C) = \emptyset$. This scenario also indicates that m will be invoked externally because $\mathbb{P}TestClass$ are not the real invoker of m instead they are testing the functionalities of $\mathbb{P}Class$.

As the scenarios #3 and #4 indicate that m has external connections, our proposed technique identifies and eliminates ABMMRS based on these scenarios. The formal schema of our proposed technique is given in what follows.

ABMMRS*Instance*

$$\begin{aligned} & m? : M \\ & C_{\text{caller}} : \mathbb{P}Class \\ & T_{\text{caller}} : \mathbb{P}unittestsClass \\ & R_C : m? \nrightarrow C_{\text{caller}} \\ & R_T : m? \nrightarrow T_{\text{caller}} \\ & Is_{\text{ABMMRS}}! : \text{ABMMRS} \end{aligned}$$

$$Is_{\text{ABMMRS}}! = true \Rightarrow (ran\ R_C = \varnothing \wedge ran\ R_T = \varnothing) \vee (ran\ R_C = \varnothing \wedge ran\ R_T \neq \varnothing)$$

[f]Implements the core functionality of a software system.
[g]These classes test the functionality of Regular classes unit by unit.

3.2. *Algorithm of identifying and eliminating ABMMRS*

To find the non-ABMMRSs, it is required to identify and eliminate ABMMRS from a list of MMRSs. The steps of identifying and eliminating ABMMRS are displayed in Algorithm 1. Algorithm 1 takes the source code of software systems and list of MMRS as input. Then, it provides a list of non-ABMMRS after identifying and eliminating ABMMRSs. After getting the source code of a software system, Abstract Syntax Tree (AST) is generated which contains details of the source code such as attributes, methods, classes, the invocation of methods and so on. In Algorithm 1, AST of a system S is stored in a variable A (Line #2). The other two variables *RClass* and *TClass* are declared for storing Regular and Unit test classes of S (Line #3 and #4). Unit test classes are identified using their annotations[h] from A and the rest of classes are considered Regular classes. Next, it is iterated over LMMRS (Line #5) for eliminating ABMMRS. In this loop, a verification step (Line #6 to #7) is completed for each MMRS to examine whether the MMRS is application-breaking. This verification is performed based on the Mathematical Formalization of ABMMRS. According to the formalization, an MMRS is considered as ABMMRS and removed from LMMRS (Line #8) if the suggestion is provided on a *public* method which is not used by any *RClass* or it is only used by the *TClass*. In this algorithm, *IsNotUsedByRegularClasses* examines whether a method is invoked by any *RClass*. Likewise, *IsNotUsedByTestClasses* and *IsUsedByTestClasses* verify

Algorithm 1 . Identification and Elimination of ABMMRS

Require: Source Code of a Software Systems S and List of MMRS
Ensure: List of MMRS those do not break applications after refactoring

```
1.  function eliminateABMMRS(S, LMMRS)
2.      A ← AST of S
3.      RClass ← List of Regular classes of S
4.      TClass ← List of unit test classes of S
5.      for MMRS in LMMRS do
6.          if IsPublic(MMR) then
7.              if IsNotUsedByRegularClasses(MMRS, RClass) ∧
    (IsNotUsedByTestClasses(MMRS, TClass) ∨
    IsUsedByTestClasses(MMRS, TClass)) then
8.                  LMMRS.remove(MMR)
9.              end if
10.         end if
11.     end for
12.     return LMMRS
13. end function
```

[h] The test frameworks (e.g. JUnit, NUnit and TestNG) provide this feature.

whether a method is invoked by any *TClass*. After removing all ABMMRS from LMMRS, the rest of MMRSs are returned (Line #12).

4. Implementation and Result Analysis

In order to perform comparative result analysis, the proposed ABMMRS Eradicator was implemented as a software tool (Eclipse-plugin) using Java programming language. To evaluate the effectiveness of the ABMMRS Eradicator, an empirical study was conducted on nine open-source projects (Table 1) from Github. These projects contain four library projects (jgroup, jbot, jest and greenmail) and five Spring MVC [17] projects (base spring, greenhouse, SpringBlog, SpringMVCDemo and spring-petclinic). Previous works [19, 25] in this domain inspired us to select open source projects for experiments. To select these projects, we focus on the lines of code (LOC), number of classes (NOC) and number of methods (NOM) of each project. All projects in the dataset are developed using Java. The reason is that ABMMRS Eradicator was compared with JDeodorant which only works in Java. Lastly, we design the following research question to evaluate our proposed technique:

- **Research Question:** What is the effectiveness of the proposed technique in eliminating ABMMRS?

Table 1. The dataset of this study. 1st column: Project Name, 2nd column: LOC — Lines of code in each project, 3rd column: NOC — Number of classes in each project, 4th column: NOM — Number of methods in each project.

Project	LOC	NOC	NOM
base spring	2579	44	147
greenhouse	27376	440	1703
jbot	386	36	534
jest	28290	488	1987
jgroup	103081	770	8186
SpringBlog	2241	46	185
SpringMVCDemo	457	6	53
spring-petclinic	2527	37	144
greenmail	22658	268	1386
Total	189595	2135	14325

4.1. *Environmental setup*

This section outlines the required steps for conducting the experimental analysis of the proposed technique named ABMMRS Eradicator. At first, we invited 20 Java developers from industries for identifying ABMMRS from the dataset manually. Each of these developers has more than five years of experience in Java development (Spring MVC and libraries). The developers participated in our experiment as part of

industry-academia collaboration. The reason of identifying ABMMRS manually is to compare those with the identified ABMMRS of the ABMMRS Eradicator. The developer team undertook the following steps to identify ABMMRS:

(1) At first, they generated MMRSs using JDeodorant's Eclipse Plug-in from the dataset (Table 1). The MMRSs can be found here.[i]
(2) Next, they tried to refactor (moving envied methods to the target classes) MMRS found in Step 1. First 20% of developers used JDeodorant's Eclipse Plug-in for refactoring. Then, 80% of the total developers manually verified the automated refactoring of JDeodorant's Eclipse Plug-in. The existing study [37] shows that the number of tool users is relatively low compared to manual refactoring. For this reason, we divide the developers into two groups.
(3) In the last step, they observed applications' behavior. For web-based MVC projects, the behavior is observed by compiling and running the projects. For libraries, the behavior is observed by running the unit tests. If any error was noticed after refactoring a particular MMRS, they considered it as an ABMMRS. The list of ABMMRS identified by the developer team is displayed in Table 3.

For a clear understanding, the list of ABMMRS identified by the developer team is grouped into three categories and described as follows:

(1) **ABMMRS in MVC project:** Although Action methods in MVC projects are identified as FE instances by JDeodorant due to coupling with other classes, yet the developer team could not refactor these and finally marked these methods as ABMMRS. It is observed that applying MMR on these Action methods breaks applications' behavior. The MMRSs in Table 3 with Id #1, #3, #4, #5, #6, #7, #8, #9, #10, #11, #12, #13, #16, #17, #18, #19, #20, #21, #22, #23, #24, #25 and #26 were found in this category.
(2) **ABMMRS in libraries:** It is found that the API methods are also identified as FE instances by JDeodorant for being coupled with other classes, the developer team, however, could not apply MMR on these methods. The reason is that applying MMR on these methods breaks client projects' behavior. In this study, an ABMMRS in each library was identified by determining whether it changed the behaviors of either (1) unit tests or (2) the test projects which were provided by the library itself. To this end, the developer team identified API methods in libraries by going through the API Documentation [38–41]. of each library and marked these methods as ABMMRS. For each library, a MMRS was marked as ABMMRS if it targeted an API method, and (2) all the API methods were identified using the API documentation of the library. The MMRSs with Id #27, #28, #29 and #30 are found to be ABMMRS of libraries in Table 3
(3) **Other ABMMRS:** Apart from the above two categories, JDeodorant provided few non-refactorable MMRSs which are also marked as ABMMRS by the

[i]List of MMRS identified by developers: https://bit.ly/2IUqku0.

developer team eventually. The reason behind applying refactoring these MMRSs ensues compilation and build errors, i.e. the method annotations in source class is not supported by the target class. The MMRSs with Id #2, #14 and #15 listed in Table 3 are ABMMRS of this category.

4.2. *Evaluation*

To answer the designed Research Question, ABMMRS Eradicator was applied on the dataset to eliminate ABMMRS from the list of MMRSs. Next, it is observed that to answer *Research Question* whether ABMMRS Eradicator can automatically eliminate all ABMMRS under categories A–C. Finally, we discuss in Sec. 4.3 how much the proposed technique improved the Move Method recommendations compared to an existing technique.

Answer to the Research Question: To measure the effectiveness of the proposed technique, we use Precision, Recall and F-score matrices. From Table 2, *Total MMRSs*, *Actual ABMMRS* and *ABMMRS Eliminated* are calculated for measuring Precision, Recall and *F*-score.

- **Total MMRSs:** This refers to the total number of MMRSs found on the dataset is 91. These MMRSs were collected by applying JDeodorant's Eclipse Plug-in [42] on the dataset.
- **Actual ABMMRS:** There were 30 ABMMRSs (categories A–C) out of 91 MMRSs in the dataset identified by the developer team. These MMRSs must be removed because these actually break the applications' behavior after refactoring.
- **ABMMRS Eliminated:** ABMMRS Eradicator correctly eliminated 27 ABMMRSs out of 30 Actual ABMMRSs and one ABMMRS elimination was incorrect. Basically, it correctly eliminated all ABMMRSs under Categories A and B.

Table 2. The actual and identified ABMMRS. 1st column: Project Name, 2nd column: MMRS — Number of MMRSs provided by JDeodorant, 3rd column: ABMMRS — Number of actual ABMMRS identified by developers, 4th column: Eliminated ABMMRS — Number of ABMMRS eliminated by the proposed technique from 3rd Column.

Project	MMRS	ABMMRS	Eliminated ABMMRS
base spring	7	3	2
greenhouse	18	10	10
jbot	2	1	1
jest	1	1	1
jgroup	33	2	2
SpringBlog	8	6	4
SpringMVCDemo	4	4	4
spring-petclinic	4	3	3
greenmail	14	0	1
Total	91	30	28

Table 3. Application-breaking Move Method suggestions identified by developers.

Id	Project	Method	Source	Target
1	base spring	sendUpdate - PasswordEmail	UpdatePassword -Controller	UserDAO
2	base spring	createUrl	EmailService	domain.User
3	base spring	register	RegistrationController	UserDAO
4	greenhouse	sendInvites	MailInviteController	MailInviteForm
5	greenhouse	welcomeMail	WelcomeMail -Transformer	Account
6	greenhouse	changePassword	ResetPassword -Controller	Change-PasswordForm
7	greenhouse	settingsPage	SettingsController	Account
8	greenhouse	updateEvent	EventLoadController	NFJSLoader
9	greenhouse	create	AppController	Account
10	greenhouse	view	AppController	Account
11	greenhouse	delete	AppController	Account
12	greenhouse	editForm	AppController	Account
13	greenhouse	update	AppController	Account
14	SpringBlog	createPost	PostService	Post
15	SpringBlog	updatePost	services.PostService	Post
16	SpringBlog	show	PostController	PostService
17	SpringBlog	about	HomeController	PostService
18	SpringBlog	updateSettings	AdminController	SettingsForm
19	SpringBlog	updateSettings	AdminController	AppSetting
20	SpringMVCDemo	addBlogPost	BlogController	BlogEntity
21	SpringMVCDemo	updateBlogP	BlogController	BlogEntity
22	SpringMVCDemo	addUserPost	MainController	UserEntity
23	SpringMVCDemo	updateUserPost	MainController	UserEntity
24	spring-petclinic	initCreationForm	PetController	Owner
25	spring-petclinic	processCreationForm	PetController	Owner
26	spring-petclinic	processUpdateForm	PetController	Owner
27	jgroup	down	Channel	Event
28	jgroup	put	AWSAuthConnection	S3Object
29	jbot	reply	Bot	Message
30	jest	getObject	JestClientFactory	DroidClientConfig

As Category C does not contain any Action or API methods, ABMMRS Eradicator could not eliminate any ABMMRS from this category. In addition to that, ABMMRS Eradicator treats Unused methods [43] as ABMMRS due to the common characteristics between ABMMRS and Unused Methods, i.e. no internal or external callers/users of methods. For this reason, one ABMMRS elimination was incorrect.

$$Precision = \frac{Correctly\ Eliminated\ \text{ABMMRS}}{Total\ \text{ABMMRS}\ Eliminated} = \frac{27}{28} = 96.4\%, \tag{2}$$

$$Recall = \frac{Correctly\ Eliminated\ \text{ABMMRS}}{Total\ Actual\ \text{ABMMRS}} = \frac{27}{30} = 90\%, \tag{3}$$

$$F\text{-}score = 2 * \frac{Precision * Recall}{Precision + Recall} = 2 * \frac{0.964 * 0.90}{0.964 + 0.90} = 93.1\%. \tag{4}$$

From Eqs. (2)–(4), the Precision, Recall and *F*-score are calculated and these equations yield 96.4%, 90% and 93.1% as the Precision, Recall and *F*-score of

proposed technique, respectively. The proposed technique achieved 96.4% Precision as there were very few Unused Methods, i.e. the selected dataset has only one Unused Method named *consumeLong*.[j] Precision could have been less than the current 96.4% if there were many Unused Methods in the dataset. However, Unused Methods are another type of Code Smell that should be eliminated [43] by the developers. It is worth noting that our dataset contains less Unused Methods because most of those methods were eliminated in advance by the developers of each project. Hence, it can be said that the Precision of the proposed technique will be consistent in the presence of less Unused Methods. The current Recall 90% of the proposed technique can deteriorate in the presence of many ABMMRS under Category C (Other ABMMRS). This category was beyond the consideration of this study as it does not contain either Action or API methods. Moreover, it was assumed that JDeodorant already resolved this issue by examining compilation preconditions before providing any MMRS. In future, this issue can be solved by strengthening the compilation preconditions of JDeodorant.

4.3. *Findings and result discussion*

In this section, we first discuss the percentage of ABMMRS provided by JDeodorant for the selected dataset. Then, we represent how many of JDeodorant's ABMMRSs is eliminated by the proposed technique.

From Table 2, JDeodorant provided a total of 91 MMRSs. Among these suggestions, the developer team identified 30 out of 91 MMRSs as ABMMRSs which indicates 33.33% of JDeodorant's MMRSs broke the applications' behavior after refactoring. The percentage of ABMMRS in JDeodorant's MMRSs in each project can be measured by comparing 2nd and 3rd columns of Table 3. From there, it can be seen that 100% of JDeodorant's MMRSs in jest-droid and SpringMVCDemo were ABMMRS. The reason is that there were no FE instances except API or Action methods in jest-droid and SpringMVCDemo. Apart from this, 75%, 75%, 55.56%, 50%, 42.86%, 6.06% and 0% of JDeodorant's MMRSs were found to be ABMMRS, respectively, in spring-petclinic, SpringBlog, greenhouse, jbot, base spring, jgroup and greenmail. This amount of ABMMRSs is not trivial in the software industry because it might hamper the productivity of software development. Thus, it is required to check how many of these ABMMRSs can be eliminated automatically using the proposed ABMMRS Eradicator.

The ABMMRS Eradicator can be an appropriate choice to avoid ABMMRSs in recommending MMRSs in web-based MVC projects and libraries. By comparing 3rd and 4th columns of Table 3, it can be seen that ABMMRS Eradicator eliminated 100% ABMMRS of JDeodorant in greenhouse, jbot, jest-droid, jgroup, SpringMVCDemo and spring-petclinic. In other projects, base spring and SpringBlog, ABMMRS Eradicator eliminated 66.67% ABMMRS of JDeodorant. The rest 33.33%

[j] It is located in greenmail project of our dataset. Its source class is *CommandParser*.

ABMMRS of JDeodorant was under Category C and for this reason, ABMMRS Eradicator could not eliminate those. The overall ABMMRS elimination rate of ABMMRS Eradicator is 90% (27 out of 30). In other words, it could be said that ABMMRS Eradicator provided 90% better results than JDeodorant in recommending MMRSs for the studied dataset.

5. Threats to Validity

In this section, some factors are discussed which may jeopardize the external and internal validity of this research.

External Validity: In this study, ABMMRS was identified by a developer team rather than the actual developer of the open-source projects in dataset. Usually, these open-source projects are often contributed by so many developers continuously, it is very difficult to get the ABMMRS identified by actual developers. Thus, it is recommended that whoever identifies the ABMMRS must follow the same steps to replicate this study, otherwise, results may vary. Secondly, the identification of ABMMRS by the developer team depended on the MMRSs by JDeodorant. As a result, the failure of JDeodorant to find MMRSs to be ABMMRS would affect the result as well.

Internal Validity: As discussed earlier, the proposed technique considers Unused Methods as ABMMRS although these are not Application-breaking. Thus, the presence of many Unused Methods can affect the Precision of ABMMRS Eradicator. However, it is possible to achieve better Precision by eliminating Unused Methods in advance. The Recall of the proposed technique can also vary in the presence of many ABMMRS under Category C. This threat can be avoided by strengthening the compilation preconditions of JDeodorant.

6. Conclusion and Future Work

To increase productivity in software development, developers prefer automated techniques for identifying MMRSs from source code. However, existing techniques have a common shortcoming — applying MMRSs on Action and API methods breaks web-based MVC projects and libraries, which violates the fundamental principles of refactoring.

In this paper, a technique has been proposed that automatically finds ABMMRSs to solve this problem. The technique obtains initial MMRSs using an existing technique. For each of the suggestions, it constructs AST to check whether the suggestion is Application-breaking. This check is done by adopting two rules — whether the respective suggestion has not been used in any Regular or unit tests class, and whether the suggestion is used only in unit test classes. If the suggestion satisfies any of the rules, it is marked as Application-breaking.

For experimental analysis of the technique, a tool named ABMMRS Eradicator has been developed. The tool was run on nine open-source projects. One of the most popular tools named JDeodorant was used to obtain initial MMRSs and then, ABMMRS Eradicator was applied. The result analysis shows that the proposed technique achieved 96.4% Precision, 90% Recall and 93.1% *F*-score in detecting ABMMRSs. As mentioned previously, it has been seen that the technique fails only for Unused Methods. However, it is another code smell that should be removed from the source code and then the technique will provide more accurate results. In future, a tool will be developed that will first remove Unused Methods and then eliminate ABMMRSs.

Acknowledgments

This research is funded by the Bangladesh University Grants Commission.

References

1. F. Khomh, M. D. Penta and Y. Guéhéneuc, An exploratory study of the impact of code smells on software change-proneness, in *Proc. 16th Working Conf. Reverse Engineering*, eds. A. Zaidman, G. Antoniol and S. Ducasse (IEEE Computer Society, 2009), pp. 75–84.
2. M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice — Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems* (Springer, 2006).
3. A. F. Yamashita and L. Moonen, Do code smells reflect important maintainability aspects?, in *Proc. 28th IEEE Int. Conf. Software Maintenance* (IEEE Computer Society, 2012), pp. 306–315.
4. M. Fowler and K. Beck, *Refactoring: Improving the Design of Existing Code* (Addison-Wesley Professional, 1999).
5. M. D'Ambros, A. Bacchelli and M. Lanza, On the impact of design flaws on software defects, in *Proc. 10th Int. Conf. Quality Software*, eds. J. Wang, W. K. Chan and F. Kuo (IEEE Computer Society, 2010), pp. 23–31.
6. D. I. Sjøberg, A. Yamashita, B. C. Anda, A. Mockus and T. Dybå, Quantifying the effect of code smells on maintenance effort, *IEEE Trans. Softw. Eng.* **39**(8) (2013) 1144–1156.
7. V. R. Basili, L. C. Briand and W. L. Melo, A validation of object-oriented design metrics as quality indicators, *IEEE Trans. Softw. Eng.* **22**(10) (1996) 751–761.
8. L. C. Briand, J. Wüst, S. V. Ikonomovski and H. Lounis, Investigating quality factors in object-oriented designs: An industrial case study, in *Proc. Int. Conf. Software Engineering*, eds. B. W. Boehm, D. Garlan and J. Kramer (ACM, 1999), pp. 345–354.
9. S. R. Chidamber, D. P. Darcy and C. F. Kemerer, Managerial use of metrics for object-oriented software: An exploratory analysis, *IEEE Trans. Softw. Eng.* **24**(8) (1998) 629–639.
10. L. C. Briand, J. Wüst and H. Lounis, Using coupling measurement for impact analysis in object-oriented systems, in *Proc. Int. Conf. Software Maintenance* (IEEE Computer Society, 1999), pp. 475–482.
11. M. A. Chaumun, H. Kabaili, R. K. Keller, F. Lustman and G. Saint-Denis, Design properties and object-oriented software changeability, in *Proc. 4th European Conf. Software Maintenance and Reengineering* (IEEE Computer Society, 2000), pp. 45–54.

12. J. Galloway, P. Haack, B. Wilson and K. S. Allen, *Professional ASP. NET MVC 4* (John Wiley & Sons, 2012).
13. R. Johnson, J2EE development frameworks, *Computer* **38**(1) (2005) 107–110.
14. W. Cui, L. Huang, L. Liang and J. Li, The research of PHP development framework based on MVC pattern, in *Proc. 14th Int. Conf. Computer Sciences and Convergence Information Technology* (IEEE, 2009), pp. 947–949.
15. GitHub, Library, 2018, https://github.com/topics/library.
16. T. Mens and T. Tourwé, A survey of software refactoring, *IEEE Trans. Softw. Eng.* **30**(2) (2004) 126–139.
17. P. Software, Spring Framework, 2018, https://github.com/spring-projects/spring-framework.
18. Microsoft, ASP.NET MVC Overview, 2018, https://msdn.microsoft.com/en-us/library/dd381412(v=vs.108).aspx.
19. N. Tsantalis and A. Chatzigeorgiou, Identification of move method refactoring opportunities, *IEEE Trans. Softw. Eng.* **35**(3) (2009) 347–367.
20. F. Simon, F. Steinbrückner and C. Lewerentz, Metrics based refactoring, in *Proc. 5th Conf. Software Maintenance and Reengineering*, eds. P. Sousa and J. Ebert (IEEE Computer Society, 2001), pp. 30–38.
21. J. A. Dallal, Predicting move method refactoring opportunities in object-oriented code, *Inform. Software Tech.* **92** (2017) 105–120.
22. V. Sales, R. Terra, L. F. Miranda and M. T. Valente, Recommending move method refactorings using dependency sets, in *Proc. 20th Working Conf. Reverse Engineering*, eds. R. Lämmel, R. Oliveto and R. Robbes (IEEE Computer Society, 2013), pp. 232–241.
23. F. Palomba, A. Panichella, A. D. Lucia, R. Oliveto and A. Zaidman, A textual-based technique for smell detection, in *Proc. 24th IEEE Int. Conf. Program Comprehension*, (IEEE Computer Society, 2016), pp. 1–10.
24. F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, A. D. Lucia and D. Poshyvanyk, Detecting bad smells in source code using change history information, in *Proc. 28th IEEE/ACM Int. Conf. Automated Software Engineering*, eds. E. Denney, T. Bultan and A. Zeller (IEEE, 2013), pp. 268–278.
25. G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk and A. De Lucia, Methodbook: Recommending move method refactorings via relational topic models, *IEEE Trans Softw. Eng.* **40**(7) (2014) 671–694.
26. A. Majeed and I. Rauf, MVC architecture: A detailed insight to the modern web applications development, *Peer Rev. J. Solar Photoenergy Syst.* **1**(1) (2018) 1–7.
27. A. J. Riel, *Object-Oriented Design Heuristics* (Addison-Wesley Publishing Company, 1996).
28. D. A. Aruliah, C. T. Brown, N. P. C. Hong, M. Davis, R. T. Guy, S. H. D. Haddock, K. Huff, I. Mitchell, M. D. Plumbley, B. Waugh, E. P. White, G. Wilson and P. Wilson, Best practices for scientific computing, preprint (2012), arXiv:1210.0530.
29. W. F. Opdyke, *Refactoring Object-Oriented Frameworks* (University of Illinois at Urbana-Champaign, 1992).
30. R. Sokal and P. Sneath, *Principles of Numerical Taxonomy* (San Francisco, CA, W. H Freeman, 1963).
31. P. H. Sneath, R. R. Sokal *et al.*, *Numerical Taxonomy. The Principles and Practice of Numerical Classification* (Taylor & Francis, Ltd., 1973).
32. R. M. Masudur, R. R. Rubby, K. S. Mostafa, S. Abdus and R. M. Rayhanur, MMRUC3: A recommendation approach of move method refactoring using coupling, cohesion, and contextual similarity to enhance software design, *Softw. Pract. Exper.* **48**(9) (2018) 1–28.

33. J. Chang and D. Blei, Relational topic models for document networks, in *Proc. 12th Int. Conf. Artificial Intelligence and Statistics*, 2009, pp. 81–88.
34. J. Woodcock and J. Davies, *Using Z: Specification, Refinement, and Proof* (Prentice Hall, Englewood Cliffs, 1996).
35. Tutorialspoint, Spring MVC Framework Annotation, 2018, https://bit.ly/37Hs7Mg.
36. Microsoft, ActionResult Class, 2018, https://msdn.microsoft.com/en-us/library/system.web.mvc.actionresult(v=vs.98).aspx.
37. E. Murphy-Hill, C. Parnin and A. P. Black, How we refactor, and how we know it, *IEEE Trans. Softw. Eng.* **38**(1) (2011) 5–18.
38. jgroups.com, API Documentation of JGroup, 2018, http://www.jgroups.org/.
39. searchbox io, API Documentation of Jest, 2018, https://github.com/searchbox-io/Jest/tree/master/jest#usage.
40. icegreen.com, API Documentation of GreenMail, 2018, http://www.icegreen.com/greenmail/javadocs/index.html.
41. ramswaroop, API Documentation of JBot, 2018, https://github.com/ramswaroop/jbot#why-use-jbot.
42. N. Tsantalis, JDeodorant Eclipse Plug-in, 2018, https://marketplace.eclipse.org/content/jdeodorant.
43. S. R. Nair and S. M. Lobo, Efficient dead code elimination, US Patent 7,353,503, (2008).