2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering

Domino Effect: Move More Methods Once A Method Is Moved

Hui Liu, Yuting Wu, Wenmei Liu, Qiurong Liu, and Chao Li
School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China
E-mail: Liuhui08@bit.edu.cn, wuyuting@bit.edu.cn, liuwenmei@bit.edu.cn, liuqiurong@bit.edu.cn, 2276494689@qq.com

***Abstract*—Software refactoring is a popular technology to improve the design of existing source code, and thus it is widely used to facilitate software evolution. Moving methods is one of the most popular refactorings. It helps to reduce coupling between classes and to improve cohesion of involved classes. However, it is difficult to manually identify such methods that should be moved. Consequently, a number of approaches and tools have been proposed to identify such methods based on source code metrics, change history, and textual information. In this paper we propose a new way to identify methods that should be moved. Whenever a method is moved, the approach checks other methods within the same class, and suggests to move the one with the greatest similarity and strongest relationship with the moved method. The rational is that similar and closely related methods should be moved together. The approach has been evaluated on open-source applications by comparing the recommended move method refactorings against refactoring histories of the involved applications. Our evaluation results show that the approach is accurate in recommending methods to be moved (average precision 76%) and in recommending destinations for such methods (average precision 83%). Our evaluation results also show that for a substantial percentage (27%) of move method refactorings, the proposed approach succeeds in identifying additional refactoring opportunities.**

I. Introduction

Software refactoring is to improve software quality by restructuring the internal structure of software applications whereas their external behaviors are not changed [1], [2]. As an effective way to improve the design of existing source code [3], software refactoring is popular [4], [5], [6]. It is widely used to facilitate software evolution as well [2]. To this end most of the modern IDEs contain built-in refactoring tools because tool support is crucial for the success of software refactoring [7], [8], [9]. These tools provide good support for fine-grained refactorings, e.g., the movement of a single method (move method refactoring). They ensure the completeness of such fine-grained refactorings. For example, when a method is moved from class A to class B, refactoring tools will update all references to the method to avoid any syntactic or semantical errors.

Moving methods is one of the most popular refactorings. As Fowler states [3, Chapter 7], moving methods is "*the bread and butter of refactoring*". Moving methods are often used to resolve code smells of *feature envy*. If a method is more interested in another class than the class it is defined, we call it a feature envy. In this case, the method had better be moved to the class that it is interested in. The benefits of such refactorings, i.e., moving methods, are twofold. First, moving methods reduces coupling between classes. Reduction in coupling is beneficial because high coupling is often associated with lower productivity and greater evolution cost [10], [11]. Second, moving methods (functions) to classes that logically provide such functions straightens out the business logic of the applications, and thus improves the readability and maintainability of such applications.

However, it is difficult to manually identify methods that should be moved [12], [13]. Consequently, a number of approaches have been proposed to identify such methods automatically or semi-automatically [14]. Most of such approaches relies on metrics, e.g., coupling [15], [12], [16] , some of them exploit textual information [14], and some of them take advantage of evolution histories of source code [13]. However, to the best of our knowledge, none of them has exploited move method refactorings that have been conducted recently.

To this end, in this paper we propose a new approach to identify moving method opportunities based on conducted move method refactorings. Whenever a method *m* is moved from c_s (source class) to another class c_d (destination class), the approach looks for methods within the source class that may deserve a movement to the destination class c_d. The rational for the approach is that if two methods are strongly coupled and closely related in business logic, when one of them is moved the other may deserve a movement as well.

The proposed approach is evaluated on open-source applications. Among the 191 methods that the approach suggests to move, 146 (76.44%) have been actually moved in the following versions of such applications. 11 of the recommended methods have been removed from the following versions and thus the correctness of such recommendations cannot be decided. 7 of the recommendations represent true move method opportunities that have not yet been conducted. Consequently, the actual precision of the approach is up to 86%=(146+7)/(191-11). Among the 146 recommended methods that have been actually moved, 121 (83%) have been moved to the destination classes that are suggested by the approach. Our evaluation results also show that the proposed approach succeeds in expanding a substantial percentage (27%) of the move method refactorings. For each of them, at least one (2.2 on average) moving method opportunity is accepted, i.e., the recommended refactoring has been actually conducted in following versions. As a conclusion, the proposed approach is accurate in identifying methods that should be

978-1-5090-1855-0/16 $31.00 © 2016 IEEE
DOI 10.1109/SANER.2016.14

moved and recommending destinations for the movement.

This paper offers the following contributions:

- First, we propose a new way to identify move method refactoring opportunities. Compared to existing approaches that identify move method refactoring opportunities, the main advantage of the proposed approach is that it exploits conducted move method refactorings.
- Second, the proposed approach has been evaluated on four open-source applications. Our evaluation results show that the proposed approach is accurate. 76% of recommended methods have been actually moved later on, and 83% of such methods are moved to the recommended destinations.

The rest of the paper is structured as follows. Section II presents a short review of related research. Section III proposes the approach to identify methods that should be moved. Section IV presents an evaluation of the proposed approach on open-source applications. Section V discusses related issues. Section VI provides conclusions and potential future work.

II. Related Work

A. Identification of Move Method Opportunities

As stated by Bavota et al. [14], there are only a small number of approaches that could identify move method refactoring opportunities. The first one was proposed by Simon et al. in 2001 [15]. They proposed a distance based cohesion to measure how closely two entities (methods or attributes) are related:

$$distance(e_1, e_2) = 1 - \frac{|p(e_1) \cap p(e_2)|}{|p(e_1) \cup p(e_2)|} \quad (1)$$

where e_1 and e_2 are two software entities, and $p(e)$ is the set of properties that are possessed by e. If e is a method, $p(e)$ includes e itself, all methods that are directly invoked by e, and all attributes that are directly accessed by e. If e is an attribute, $p(e)$ includes e itself and all methods that directly access e. As a conclusion, the distance between two entities is computed based on the collection of their shared properties. With the distance metrics, Simon et al. draw entities on a graph by a layout algorithm, and the geometric distances between entities correspond to the distance calculated by Formula 1. On such a graph, developers may visually identify methods that should be moved: if a method is closer to entities of another class than those of its enclosing class, the method had better be moved.

The second approach was proposed by Seng et al. [16]. The approach searches for the optimal class structure with the fitness function defined as follows:

$$fitness(S) = \sum_{i=1}^{n} w_i * \frac{M_i(S) - M_{init_i}(s)}{M_{max_i}(S) - M_{init_i}(s)} \quad (2)$$

where S is the application to be refactored, M(s) is a vector composed of seven metrics: weighted method count, response for class, information-flow-based coupling, tight class cohesion, information-flow-base-cohesion, lack of cohesion, and stability. $M_{init_i}(s)$ is the initial value of the metrics, and $M_{max_i}(S)$ is the maximal values obtained by a calibration run optimizing each metric alone beforehand. Each genotype in the search algorithm contains a sequence of refactorings, e.g., move method refactorings. By searching for the optimal genotype that maximizes $fitness(S)$, the approach identifies a sequence of refactorings that lead to the optimal class structure of the involved application. To the best of our knowledge, this approach is the first search-based approach to identifying move method refactoring opportunities.

The third approach was proposed by Tsantalis and Chatzigeorgiou [12]. They measure the distance between method e and class C with the following formula when e does not belong to C:

$$distance(e, C) = 1 - \frac{S_e \cap S_C}{S_e \cup S_C},\ where\ S_C = \bigcup_{e_i \in C} \{e_i\} \quad (3)$$

Otherwise, the distance is computed as follows:

$$distance(e, C) = 1 - \frac{S_e \cap S'_C}{S_e \cup S'_C},\ where\ S'_C = S_C \backslash \{e_i\} \quad (4)$$

With this distance, method e is suggested to be moved to class C_{target} that has the shortest distance to the method if the movement satisfies some preconditions. Such preconditions ensure that the movement will not change the external behaviors of the involved application. To the best of our knowledge, it is the first approach to identifying move method refactorings based on the distance between methods and classes. The distance is different from that defined by Simon et al. [15] since the latter measures the distance between two methods (or attributes) whereas the distance defined in Formula 3 and 4 measures the distance between a method and a class.

The fourth approach was proposed by Sales et al.[17]. They suggest to move a method m to the class c that is most similar to the method concerning their dependency sets. The rational of their approach is that methods in well designed classes should depend on similar types.

The fifth approach was proposed by Bavota et al. [14]. They recommend move method refactorings via Relational Topic Models (RTM). RTM identifies friends for methods. Two methods are considered as friends if they operate on the same data structures or are related to the same features or concepts. For a given method, they exploit RTM to suggest where this method should be moved: the class that contains the highest number of *friends* of this method. The key of the approach is to compute the relationship among methods according structural similarity between methods as well as textual information (e.g., identifier names and comments) extracted from the source code. To the best of our knowledge, it is the first approach to identifying move method refactorings based on textual information besides source code structures, and their evaluation results show that such information can significantly improve the performance, i.e., precision and recall.

The sixth approach was proposed by Palomba et al. [13]. The approach identifies moving method opportunities by looking for *feature envy* smells [3]. The approach takes assumption that a method affected by feature envy changes more often

with the envied class than with the class where it is defined. Consequently, if a method m is involved in commits with methods of another class (C_{target}) significantly more often than in commits with methods of its enclosing class (C_{source}), the approach suggests to move the method m to class C_{target}. To the best of our knowledge, this is the first approach to identifying move method refactorings by mining version histories of source code.

The key difference among such approaches is that they exploit different information. Simon et al. [15] exploit the distance between methods, Seng et al. [16] exploit system-level metrics, Tsantalis and Chatzigeorgiou [12] exploit the distance between methods and classes, Bavota et al. [14] exploit textual information as well structural information, and Palomba et al.[13] exploit the version histories of source code.

However, none of the approaches introduced in preceding paragraphs exploits move method refactorings that are recently conducted by developers. To this end, in this paper we exploit such information and propose a new way to identify move method refactoring opportunities. Except for such new information, our approach follows the key rational exploited by existing approaches [15], [14], [13]: closely related methods should stand together. Our approach collaborates with existing ones instead of replacing such approaches. Developers may find out move method refactorings with existing approaches (or manually), conduct the refactorings, and our approach may identify additional move method refactoring opportunities.

B. Identification of Refactoring Opportunities

There are dozens of popular refactorings besides moving methods, and there are a number of approaches and tools to identify opportunities for such refactorings [18], [19], [20], [21], [**?**]. Detailed literature review and analysis on such approaches have been made by Zhang et al. [18] and Dallal [22]. Most of such approaches are based on metrics [23], [24]. Heuristic algorithms are also frequently exploited by such approaches, e.g., the approaches by Tsantalis and Chatzigeorgiou [25], [20], approaches by Bavota et al. [26], [27], and approaches by Liu et al. [28].

Our approach differs from such approaches in that our approach identifies move method opportunities whereas such approaches identify refactoring opportunities belonging to other categories, e.g., extract method refactoring and extract class refactoring.

C. Recovery of refactorings

Recovery of refactorings serves as a basis for the evaluation of algorithms that recommend refactoring opportunities. Demeyer et. al. [29] proposed the idea of discovering refactorings by comparing program versions. They have proposed and evaluated four heuristics for detecting refactorings that occurred between two versions. Xing et. al. [30], [31] compared class diagrams generated by reverse engineering, and identified design level refactorings. Weissgerber and Diehl [32] analyzed revisions checked in by the same commit operation. Based on such analysis, they identified potential refactoring operations and ranked such operations with clone detection. Dig et. al. [33] matched elements in different versions using shingles [34], and inferred refactorings by semantic analysis. Their approach has been implemented and the tool *Refactoring Crawler* is publicly available (http://dig.cs.illinois.edu/tools/RefactoringCrawler/). Prete et. al. [35] proposed a template-based approach to identify refactorings occurred between two program versions. The approach has been implemented and the tool *REF-FINDER* is publicly available as well (http://users.ece.utexas.edu /∼miryung/software.html).

Tools like *Refactoring Crawler* and *REF-FINDER*, can discover move method refactorings, and thus they are exploited in the evaluation of our approach in Section IV. We discover move method refactorings (as a benchmark) that have been conducted on open-source applications. By comparing our recommendations against such discovered refactorings, we decide whether such recommendations are true positives or false positives.

III. Approach

A. Overview

The approach works as follows. First, the developer moves a method m from class C_{source} to class C_{target}. We note this movement as a move method refactoring r_1. Second, the propose approach checks each method within the source class C_{source}, and computes the strength of relationship between this method and the moved one. The relationship is composed of three aspects: coupling, conceptual correlation, and feature envy. Third, the approach sorts (in descending order) all methods from class C_{source} according to strength of relationship with the moved method. Fourth, the approach recommends to move the method on the top to the target class C_{target}, and presents the suggestion to the developer. Fifth, the developer may accept the suggestion, and conduct the move method refactoring (noted as r_2). In this case, the proposed approach continues recommending additional refactoring opportunities based on both r_1 and r_2. The developer may also reject the suggestion, and thus the proposed approach stops the recommendation triggered by r_1.

The rational of the proposed approach is that closely coupled and highly similar methods should be moved at the same time. To measure the strength of the relationship between a pair of methods, we compute their coupling, conceptual correlation, and their similarity in feature envy. Such metrics are presented in the following sections.

B. Coupling

The coupling between two methods is defined as follows:

$$coupling(m_1, m_2) = \frac{|p(m_1) \cap p(m_2)|}{|p(m_1) \cup p(m_2)|} \quad (5)$$

where m_1 and m_2 are two methods, and $p(m)$ is the set of properties that are possessed by m, including m itself, all methods that are directly invoked by m, and all attributes that are directly accessed by m.

The coupling between two methods is computed based on the collection of their shared properties. It follows the distances defined by Simon et al. [15] (Equation 1) and Tsantalis and Chatzigeorgiou [12] (Equations 3 and 4). The coupling is based on the *Jaccard similarity coefficient* between two set of properties (i.e., $p(m_1)$ and $p(m_2)$) whereas the distances employed by them are based on *Jaccard distance*. The relationship between such metrics can be expressed as:

$$coupling(m_1, m_2) + distance(m_1, m_2) = 1 \quad (6)$$

C. Conceptual Correlation

High coupling among methods is not always the sufficient condition for them to stand together. Conceptual correlation often plays the key role in the distribution of methods because each location (class) represents a concept in the real world and thus it should contain only such methods that are conceptually related to it. As a result, methods within the same class are often conceptually related. This is the reason why we exploit the conceptual correlation between methods to recommend move method opportunities based on moved methods.

We compute the conceptual correlation between two methods based on identifiers within such methods, including method names, parameter names, and names of local variables. We decompose each identier name into a set of terms, noted as $terms(name)$. The decomposition is based on underscores and capital letters assuming that the names follow the popular camel case or snake case naming convention. The decomposition is simple and effective while identifier names follow such naming conventions. However, if identifier names do not follow such naming conventions, the proposed approach might not work. In this case, more complicate and smarter approaches should be used. For example, *INTT: Identifier Name Tokenisation Tool* [36] proposed by Butler et al.[37] can decompose identifier names into meaningful terms even if these names do not follow camel case naming convention.

The conceptual correlation based on method names is computed as follows:

$$correlation_{name}(m_1, m_2) = \frac{|terms(n_1) \cap terms(n_2)|}{|terms(n_1) \cup terms(n_2)|} \quad (7)$$

(8)

where n_1 and n_2 are method names of m_1 and m_2, respectively.

The conceptual correlation based on parameters and local variables is computed as follows:

$$correlation_{variable}(m_1, m_2) = \frac{|terms(vs_1) \cap terms(vs_2)|}{|terms(vs_1) \cup terms(vs_2)|}$$

where vs_i (i.e., vs_1 or vs_2) is composed of parameter names and variable names from method m_i. $terms(vs_i)$ is composed of terms contained in all of the names from vs_i.

We separate method names from other identifers (i.e., parameter names and names of local variables) because method names often contain essential conceptual information whereas such names are often short. If terms from method names are equally treated with other terms from the method, it is likely that the information conveyed by such method names would be watered down by a large number of terms from parameters and variables.

D. Feature Envy

Feature envy is one of the major reasons for moving methods [3]. Tsantalis and Chatzigeorgiou [12] identify feature envy by measuring the distance between methods and classes. If a method is closer (i.e., with smaller distance) to another class than its enclosing class, the method is associated with a potential feature envy smell. We follow the distance metrics defined in Equations 3 and 4, and define the strength of feature envy between method m and class c as follows.

$$strength_{envy}(m, c) = distance(m, ec) - distance(m, c) \quad (10)$$

where ec is the enclosing class of the method. $distance$ measures the distance between methods and classes, following the definitions introduced by Tsantalis and Chatzigeorgiou [12].

We note a move method refactoring as:

$$r =< m_{mv}, c_{src}, c_{tg} > \quad (11)$$

where m_{mv} is the method that has been moved from its enclosing class c_{src} to another class c_{tg}.

For those methods that are considered as candidates for move method refactorings based on conducted refactoring r, we compare their strength of envy to that of the moved method:

$$\begin{aligned} Envy(m_{cd}, r) = \quad & strength_{envy}(m_{cd}, c_{tg}) \\ & -strength_{envy}(m_{mv}, c_{tg}) \end{aligned} \quad (12)$$

where m_{mv} is the method that has been moved to class c_{tg} during the move method refactoring r. Method m_{cd} is a candidate method that may deserve a movement to class c_{tg}. Metrics $Envy(m_{cd}, r)$ measures how strongly method m_{cd} envies class c_{tg} compared to the moved method m_{mv}.

E. Searching for Move Method Opportunities

Based on a conducted move method refactoring $r =< m_{mv}, c_{src}, c_{tg} >$ (as defined in Equation 11), the proposed approach computes the strength of recommendation for each method (m_{cd}) from class c_{src}. The strength is computed as follows:

$$\begin{aligned} strength(m_{cd}, r) \quad = \quad & coupling(m_{cd}, m_{mv}) \\ & +correlation_{name}(m_{cd}, m_{mv}) \\ & +correlation_{variable}(m_{cd}, m_{mv}) \\ & +Envy(m_{cd}, r) \end{aligned} \quad (9)$$

(13)

We collect all methods from the source class c_{src}, and identify the one with the greatest strength of recommendation. If the strength is greater than a threshold β, we recommend to move the method from c_{src} to c_{tg}.

If the initial recommendation is accepted, and software engineers carry out the recommended move method refactoring r_2, the proposed approach tries to recommend additional move

method opportunities based on both r_1 and r_2. The recommendation triggered by r_1 stops if and only if the proposed approach fails to recommend any refactoring opportunity or a recommended opportunity is rejected.

Suppose that based on move method refactoring r_1, a sequence of refactorings $acceptedR =< r_2, r_3, \cdots, r_k >$ has been recommended and conducted. We calculate the strength of recommendation for each method from c_{src} as follows:

$$strength(m_{cd}, acceptedR) = \sum_{i=1}^{k} strength(m_{cd}, r_i)/k \quad (14)$$

Among all such method, the proposed approach would recommend the one with the greatest strength of recommendation if the strength is greater than the predefined threshold β.

IV. Evaluation

The proposed approach has been implemented and evaluated on four open-source applications.

A. Research Questions

The evaluation investigates the following research questions:

- **RQ1**: How often are the recommended move method opportunities accepted?
- **RQ2**: How often are move method refactorings successfully expanded by the proposed approach? If based on move method refactoring r, the proposed approach results in at least one accepted recommendation, we say that r is expanded successfully.
- **RQ3**: Does the proposed approach result in higher precision and/or recall compared to *JDeodorant*, a well-known refactoring recommender [12]? If yes, to what extent?

RQ1 concerns the precision of the proposed approach. The precision is important because the proposed approach would be useless if it overwhelms developers with a large number of false positives, which is a result of low precision.

RQ2 concerns the usability of the proposed approach. Only if a substantial percentage of move method refactorings can be used to recommend additional move method refactorings successfully, the proposed approach can be used frequently and thus be valuable.

RQ3 concerns the recall and precision of the proposed compared to existing tools that recommend move method opportunities as well. Such approaches and tools are introduced in Section II-A. In the evaluation *JDeodorant* is selected because it is publicly available and it is well-known.

As a conclusion, investigating these questions would reveal whether the proposed approach is really useful.

B. Subject Applications

An overview of the subject applications is presented in Table I. Each subject application is introduced as follows.

Hibernate [38] is an open-source application developed by Red Hat[1]. The purpose of this project is to provide an easy way to achieve persistence in Java. *Weka* [39] is an open-source application developed by the machine learning group at the University of Waikato. It implements in Java a set of well-known machine learning algorithms. *Derby* [40] is an Apache DB subproject sponsored by Apache Software Foundation[2]. *SQuirreL SQL Client* [41] is an open-source database client in Java. Via JDBC, it provides users a unified interface to manipulate database libraries. *Phex* [42] is a P2P Gnutella file sharing program. The client connects to the Gnutella network, and enables users to to search, download, and share documents. *Camel* [43] is an open-source integration framework based on enterprise integration patterns. The application is large and complex. Consequently, in this evaluation we only analyzed the basic module *Camel-core*.

These subject applications were selected because of the following reasons. First, all of them are open-source applications whose source code is publicly available. Second, all of them are well-known and popular. Third, these applications were developed by different developers. Finally, these applications have long evolution history, and thus it is likely that a great number of move method refactorings have been conducted on these applications.

C. Benchmark

To decide whether move method opportunities recommended by the proposed approach are correct or incorrect, we build a benchmark of move method refactorings by mining version histories of involved subject applications. Move method refactorings (noted as RS) that had been conducted on subject applications are discovered by three participants with semi-automated tool support. The three participants are master level students who major in computer science. They are familiar with software refactoring, and all of them had participated in at least one refactoring-related project before the evaluation. They exploit *RefactoringCrawler* [44], *REF-FINDER* [45] and *Eclipse* to identify such refactorings. *RefactoringCrawler* and *REF-FINDER* are effective in recovering conducted refactorings (including move method refactorings) by comparing different versions of the same applications. We exploit these tools because they are publicly available and they have been proved to be effective. *Eclipse* is used to facilitate the review of Java source code.

For each of the potential move method refactorings identified by *RefactoringCrawler* or *REF-FINDER*, all of the three participants manually check related files together. By comparing related entities in different versions, they are asked to decide whether it is a real move method refactoring or not. For example, if *RefactoringCrawler* suggests that method m_{src} in class c_{src} version v_1 was moved to class c_{tg} in version v_2 (noted as m_{tg}), they should compare the involved source code in the two versions, and decide whether m_{tg} originals from m_{src} (instead of a newly introduced or renamed method), which is well known as origin analysis [46]. They should also identify false positives that are introduced by class renamings.

[1] http://www.redhat.com/en

[2] http://www.apache.org/

TABLE I
SUBJECT APPLICATIONS

Subject Application	Domain	No. of Versions	Size (varies from version to version)
Hibernate	Persistence	113	51,276 $\sim$ 200,874 LOC
Weka	Machine Learning	70	38,890 $\sim$ 272,212 LOC
Derby	DBMS	23	258,295 $\sim$ 626,560 LOC
Phex	P2P document share	25	24,969$\sim$ 102,579 LOC
SQuirreL SQL Client	SQL	43	38,890 $\sim$ 272,212 LOC
Camel	Integration framework	34	9,125 $\sim$ 84,041 LOC

D. Process

For each subject application, we collect a list of move method refactorings that have been conducted on the application. We note them as RS. For each move method refactoring r from RS, the evaluation is conducted as follows:

1) We remove r from RS.
2) We initialize the set of accepted refactorings suggested by the approached based on r as $acpR = \{\}$.
3) If the proposed approach fails to recommend any refactoring opportunity based on r and $acpR$, the evaluation process goes on to the next conducted move method refactoring from RS. In other words, the recommendation based on r terminates.
4) If the proposed approach recommends to move method m_{mv}, we check it against move method refactorings that have been actually conducted, i.e., RS. If there is a move method refactoring r_{act} from RS that has actually moved the method (m_{mv}), we say that the recommended refactoring opportunity is accepted. In this case, we move r_{act} from RS to $acpR$, and the recommendation based on r continues by turning to the preceding stage (i.e., the third stage).
5) If the recommended method m_{mv} has been removed from the following versions, we say that the recommendation is inclusive since the method has disappeared. In this case, the approach continues to recommend the next refactoring opportunity whose strength of recommendation ranks second only to m_{mv}, and the process goes back to the fourth step.
6) If the recommended method has not been actually moved by any refactoring from RS, we say that the recommendation is rejected. In this case, the evaluation process goes on to the next conducted move method refactoring from RS.

As introduced in the preceding paragraph, a recommendation is accepted if and only if the recommended method has been actually moved in one of the following versions.

E. Measurements

To answer research question **RQ1**, we define two metrics to measure the performance of the approach. First, the precision of the proposed approach in recommending methods to be moved is defined as follows:

$$P_{method} = \frac{ntp}{ntp + nfp} \tag{15}$$

where ntp is the number of true positives (accepted recommendations), and nfp is the number of false positives (rejected recommendations).

Second, we calculate the precision of the proposed approach in recommending where methods should be moved as follows:

$$P_{where} = \frac{nds}{ntp} \tag{16}$$

where nds is the number of accepted destinations recommended by the approach. It equals to the times when the recommended methods are actually moved to the destinations (target classes) as suggested by the approach.

To answer research question **RQ2**, we measured how often the proposed approach succeeds in identifying refactoring opportunities:

$$R_{exp} = \frac{N_{succeeded}}{N_{tried}} \tag{17}$$

where N_{tried} is the number of move method refactorings that the proposed approach tries to expand, and $N_{succeeded}$ is the number of refactorings that the proposed approach expands successfully. The expansion on a conducted move method refactoring r is successful if and only if at least one of the move method opportunities suggested based on r is accepted.

For example, if based on a move method refactoring (r_1), the proposed approach successfully recommended more than one refactorings (e.g., $r_2, r_3, \ldots, r_k$). In this case, r_1 is counted in both N_{tried} and $N_{succeeded}$) whereas the recommended refactorings ($r_2, r_3, \ldots, r_k$) are not counted in N_{tried} or $N_{succeeded}$. As a result, N_{tried} is always smaller than the number of move method refactorings discovered in subject applications.

F. Calibration

As introduced in Section III-E, the proposed approach suggests to move a method only if its strength of recommendation ($strength(mcd, r)$ in Equation 13) is greater than a predefined threshold β. To achieve a balance between accepted commendations (true positives) and rejected recommendations (false positives), we calibrate the proposed approach on two open-source applications: *Hibernate* and *Weka*. The calibration on *Hibernate* and *Weka* is presented in Fig. 1, Fig. 2, and Table II.

From these figures and the table, we observe that with the increase of the threshold, the precision of the approach increases both in recommending methods to be moved (i.e, P_{method}) and in recommending destinations for such movement (i.e., P_{where}). However, the increase in precision is

TABLE II
CALIBRATION

β	True Positives (Hibernate)	False Positives (Hibernate)	True Positives (Weka)	False Positives (Weka)
0.5	104	31	38	12
0.6	98	28	33	12
0.7	92	25	31	9
0.8	89	23	27	7
0.9	79	19	27	5
1.0	74	18	23	4

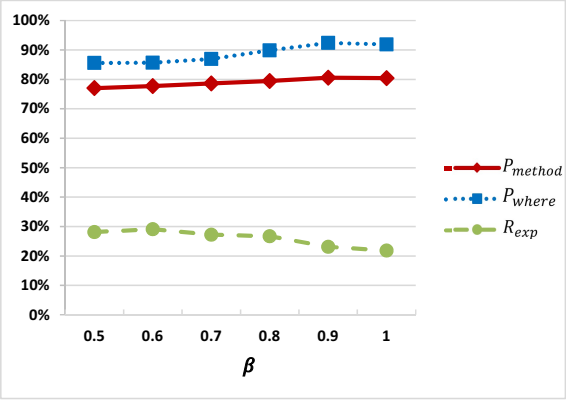

Fig. 1. Calibration on *Hibernate*

accompanied by the decrease in both the number of true positives (accepted recommendations) and R_{exp} that measures how often a conducted move method refactoring can be expanded successfully by the proposed approach.

From these figures and the table, we also observe the setting $\beta = 0.8$ achieves a good balance among different performance metrics: P_{method} is around 80%, P_{where} is around 90% and R_{exp} is greater than 25%. Consequently, the rest of the evaluation is conducted with this setting.

G. Results and Analysis

Evaluation results on four open-source applications are presented in Table III.

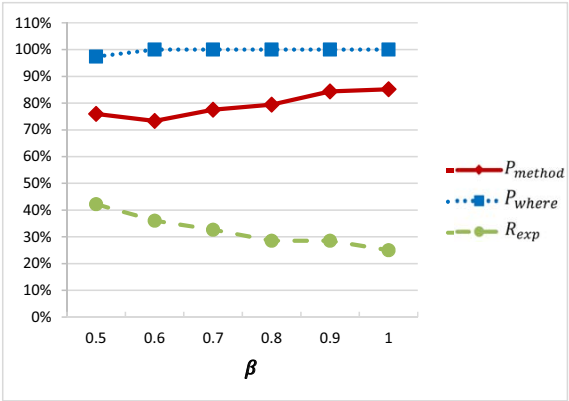

Fig. 2. Calibration on *Weka*

From the table, we observe that the precision (P_{method}) of the approach in recommending move method opportunities varies between 68% and 92%. On average 76% of the recommended move method opportunities are accepted. From these data, we conclude that the proposed approach is accurate in recommending move method opportunities (answering research question **RQ1**).

From the table we also observe that the precision (P_{where}) of the proposed approach in recommending destinations for the suggested movement varies between 67% and 100% on the subject applications. On average 83% of the recommended destinations (target classes) are correct. Form these data, we conclude that the proposed approach is accurate in recommending destinations for methods to be moved.

From the table we also observe that the proposed approach tries to recommend move method opportunities based on 244 conducted refactorings, succeeds in 67 times out of the 244 tries (27%=67/244), and generates 146 accepted move method opportunities. From this data, we conclude that a substantial number (27% on average) of move method refactorings can be expanded by the approach, i.e., leading to additional move method opportunities (answering research question **RQ2**).

H. False Positives

We manually analyze the 45 false positives, i.e., rejected recommendations. Analysis results suggest that the actual precision could be higher than that reported in Table III because of the following reasons.

First, 11 out of the 45 false positives are actually inclusive: the methods that are suggested to be moved have been removed in the following versions. As a result, we do not know whether such methods have been moved as suggested before they are finally removed. Consequently, we do not know whether such recommendations are correct or incorrect.

Second, 7 out of the 45 false positives represent true move method opportunities. In other words, they are declared wrongly as false positives because the benchmark (built in Section IV-C) is incomplete. An example of such false positives is presented in Fig. 3.

From the figure, we observe that the method *DoShutdown* is defined in class *DefaultExecutorServiceStrategy* in version 2.8.6. This method depends on another method *shutdownNow* to stop services that are still running when shutdown common is executed. In version 2.8.6, these two methods are defined in the same class.

From the figure, we also observe that the method *DoShutdown* is moved to class *DefaultExecutorServiceManager* in version 2.9.0, and this move method refactoring has been identified while benchmark was built in Section IV-C. Consequently, in the evaluation, this move method refactoring is used to trigger the proposed approach to recommend additional move method opportunities. Based on such a refactoring, the proposed approach suggests to move method *shutdownNow* from *DefaultExecutorServiceStrategy* to *DefaultExecutorServiceManager*. However, we cannot find such a move method refactoring in the benchmark, and thus the recommendation is

TABLE III
EVALUATION RESULTS

	SQuirreL SQL Client	*Phex*	*Derby*	*Camel*	Total
Discovered Move Method Refactorings (N_1)	91	125	42	132	390
Recommended Opportunities (N_2)	60	65	16	50	191
Accepted Opportunities (N_3)	55	45	12	34	146
Rejected Opportunities (N_4)	5	20	4	16	45
$P_{method} = N_3/(N_2)$	92%	69%	75%	68%	76%
Accepted Destinations (N_5)	37	39	11	34	121
$P_{where} = N_5/N_3$	67%	87%	92%	100%	83%
Expanded Refactorings (N_6)	13	22	8	24	67
Tried Refactorings (N_7)	36	80	30	98	244
$R_{exp} = N_6/N_7$	36%	28%	27%	25%	27%

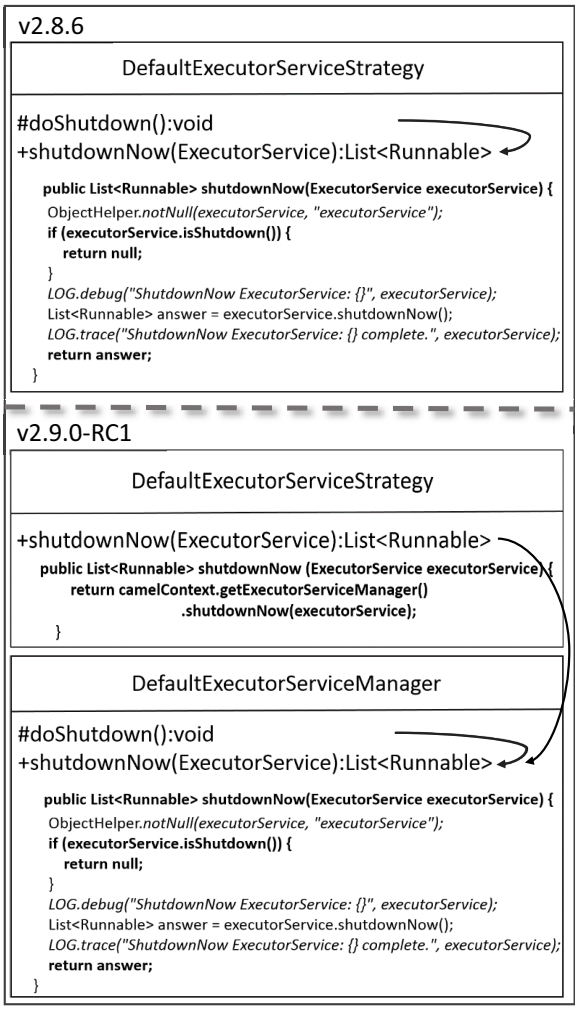

Fig. 3. False Positive Analysis: An Example from *Camel*

taken as a false positive. Although there is a method named *shutdownNow* in the target class *DefaultExecutorServiceManager*, neither *RefactoringCrawler* [44] nor *REF-FINDER* [45] decides that this method represents a move method refactoring. One possible reason is that the original class *DefaultExecutorServiceStrategy* contains a method with the same signature as well, and this method is taken as a modified version of the original method (method *shutdownNow* in version 2.8.6).

However, by comparing the related source code, we find that the method *shutdownNow* in class *DefaultExecutorServiceManager* (version 2.9.0) does represent a move method refactoring. The reasons are explained as follows.

1) The original method *shutdownNow* (in class *DefaultExecutorServiceStrategy* version 2.8.6) is identical to the method *shutdownNow* in class *DefaultExecutorServiceManager* (version 2.9.0), and it is different from method *shutdownNow* in class *DefaultExecutorServiceStrategy* (version 2.9.0).
2) In version 2.9.0, the method *shutdownNow* in class *DefaultExecutorServiceStrategy* is not called by any method. It likely that this method remains for compatibility, i.e., providing all public methods that are provided in elder versions.
3) In the original version (2.8.6), method *DoShutdown* (in class *DefaultExecutorServiceStrategy*) depends on the method *shutdownNow* in class *DefaultExecutorServiceStrategy* to terminate services. However, when this method is moved to class *DefaultExecutorServiceManager* (version 2.9.0), it depends on method *shutdownNow* in class *DefaultExecutorServiceManager* to stop such services instead of the method from *DefaultExecutorServiceStrategy*.
4) In version 2.9.0, the method *shutdownNow* in class *DefaultExecutorServiceStrategy* does nothing but calling *shutdownNow* from class *DefaultExecutorServiceManager* to stop services.

Based on such analysis, we conclude that the method *shutdownNow* has been moved from class *DefaultExecutorServiceStrategy* to *DefaultExecutorServiceManager*, but a new method with the name signature is added for compatibility. Consequently, the recommendation generated by the proposed approach, i.e., moving method *shutdownNow* from class *DefaultExecutorServiceStrategy* to *DefaultExecutorServiceMan-*

TABLE IV
EVALUATION RESULTS OF *JDeodorant*

	SQuirreL SQL Client	*Phex*	*Derby*	*Camel*	Total
Discovered Move Method Refactorings	91	125	42	132	390
Recommended Opportunities	373	272	556	156	1357
Accepted Opportunities	1	3	0	5	9
Accepted Destinations	0	0	0	0	0

ager, is not really a false positive (rejected recommendation). It should be taken as a true positive.

We conclude from the analysis that the actual precision of the proposed approach is up to $86\% = (146+7)/(191-11)$.

*I. Investigating Research Question **RQ3***

As introduced in Section II-A, besides our approach there are a number of approaches and tools that can identify move method opportunities. However, none of them takes advantage of conducted move method refactorings as we do. Consequently, it would be nonsense to compare the performance of our approach against any of them since our approach requires additional information. However, it is interesting and it makes sense to investigate to what extent by taking more information, i.e., conducted move method refactorings, can improve the performance (precision and recall).

In the evaluation, we select *JDeodorant* [12] as the representative and state of the art approach that does not take advantages of conducted move method refactorings. *JDeodorant* is selected because it is publicly available[3] and it is well-known [14].

We apply *JDeodorant* to such versions of the involved subjects (i.e., *SQuirreL SQL Client*, *Phex* , *Derby* , and *Camel*) that contain at least one move method refactoring from the benchmark built in Section IV-C. The results are presented in Table IV. The first two rows (including the row titles) of Table IV are identical to those of Table III, showing the number of move method refactorings discovered in such applications. The third row shows the number of move method opportunities suggested by *JDeodorant*, excluding duplicate suggestions that are generated because *JDeodorant* checks multi-versions of the same applications. The fourth row shows the number of suggested move method refactorings that have been actually conducted according to the benchmark built in Section IV-C (i.e. accepted opportunities). The fifth row shows the number of methods that have been moved to the target classes as suggested by *JDeodorant*.

By comparing Table IV and Table III, we observe that our approach (taking more information) leads to more accepted move method opportunities and higher precision. It should be noted that the precision of *JDeodorant* is lower than that reported in [14] where 15% of *JDeodorant*'s suggestions get positive answers to "*would you apply the proposed refactoring?*" One reason for the difference is that the benchmark built in Section IV-C may be incomplete, i.e. some conducted move method refactorings may be missed. Another reason for the difference is that developers may declare that they would like to move a method (giving a positive answer) that has not yet been actually moved. In other words, the fact that a method has not been moved does not necessarily suggest that the developer does not want to move it. However, even the higher precision (15%) of *JDeodorant* [14] is still lower than that of our approach.

We also analyze the nine accepted move method opportunities recommended by *JDeodorant*. We find that three of them have been missed by our approach. Such opportunities are missed by our approach because they are lonely: there are no move method refactorings that are similar to them. Our approach depends on similarity between refactorings, and thus it cannot identify lonely refactoring opportunities. In contrast, existing approaches, e.g., *JDeodorant* does not depend on such similarity and thus they may be able to identify such lonely opportunities.

We conclude from such results that taking conducted move method refactorings into consideration we can improve the recall and precision in recommending move method opportunities (answering research question **RQ3**). We also conclude from such results that our approach and existing approaches complement one another. Developers may identify move method opportunities with *JDeodorant* or other similar tools (e.g., *MethodBook*), conduct such refactorings, and our approach may suggest additional move method opportunities.

J. Threats to Validity

A threat to external validity comes from the limited number of subject applications. The proposed approach is calibrated on two applications, and then is evaluated on four applications. Since the number of subject applications is small, special characteristics of such applications may have biased the evaluation results. Consequently, the conclusions drawn on such applications may not be generalized to other applications. Only a small number of subject applications is involved because it is difficult and time consuming to build the benchmark on open-source applications, i.e., identifying all move method refactorings by mining version histories of such applications as introduced in SectionIV-C. To reduce the threat, we select subject applications that are developed by different developers and belong to different domains. Such applications are also well-recognized and open-source. However, to reduce the threat, further evaluation of the proposed approach should be conducted on more applications in the future.

A threat to construct validity comes from the inaccuracy in benchmarks built in SectionIV-C. To distinguish true positives from false positives, we build the benchmarks by discovering move method refactorings that have been actually conducted on the subject applications. However, the identification of conducted refactorings may be inaccurate, and thus the measurements in Section IV-E might be incorrectly

[3]http://users.encs.concordia.ca/∼nikolaos/jdeodorant/

calculated. The reasons for the inaccuracy in benchmarks are explained as follows. First, *RefactoringCrawler* and *REF-FINDER* may be inaccurate in identifying move method refactorings. Conducted move method refactorings are identified by *RefactoringCrawler* and *REF-FINDER* that may report false positives and may miss some refactorings (false negatives). As suggested by Soetens et al. [47], such snapshot based tools may miss refactorings that are masked by other editing activities. As a result, the resulting benchmarks may be inaccurate. To reduce the threat, participants manually checked files that *RefactoringCrawler* or *REF-FINDER* reported to contain potential move method refactorings. However, manual checking might miss refactorings as well. For example, the manual check may miss refactorings that are conducted on files where *RefactoringCrawler* and *REF-FINDER* fail to identify any move method refactorings. The second reason for the inaccuracy in benchmarks is that the move method refactorings reported by *RefactoringCrawler* and *REF-FINDER* are rechecked by outsiders (students) instead of the original developers of the subject applications. As a result, the manual check might be inaccurate because of lack of system knowledge and subjectiveness and because of the large amount of potential refactorings reported by *RefactoringCrawler* and *REF-FINDER*. To improve the recall, precision of such tools are reduced, which results in a large number of false positives. For example, the precision of *RefactoringCrawler* is less than one percentage. To reduce the threat, three participants check all of the potential refactorings together. In case of diverging decisions, they discussed and voted if needed before the final decision is made. In the future, it would be interesting to evaluate the proposed approach on applications where refactorings have been exactly recorded. On such applications, the benchmarks are built automatically and accurately.

Another threat to construct validity is that we compare recommended move method opportunities against refactoring histories. The fact that a method has not been moved (and thus not involved in refactoring histories) does not necessarily suggest that the method should not be moved. It is possible that the developers have missed such refactoring opportunities, and they may conduct such refactorings once the suggestions are presented to them.

V. Discussion

A. Synonyms and Abbreviation

The proposed approach calculates the similarity between two identifier names by counting the common terms appearing in both names. However, it does not take synonyms or abbreviations into consideration. Consequently, the resulting similarity may be inaccurate. For example, the proposed approach takes 'app' and 'application' as two totally different terms although the former is an abbreviation of the latter.

In future, such limitations may be resolved by introducing dictionaries, e.g., *WordNet*[4]. By consulting such dictionaries, the approach may recognize some synonyms and abbreviations. As a result, the calculation of text similarity may be improved. However, both synonym analysis and abbreviation analysis might complicate the proposed approach and make it hard to understand or implement. Synonym analysis and abbreviation analysis in source code are difficult and complex because terms (and abbreviations) usually have different meaning in different contexts and thus whether two terms are synonymous depends on their contexts. However, context analysis for terms in short identifier names is difficult.

[4]http://wordnet.princeton.edu/

B. Subjective Evaluation

The proposed approach has been evaluated on the histories of open-source applications (Section IV). The objective and quantitative evaluation results are promising, suggesting that the proposed approach is accurate and useful. However, no subjective evaluation of the approach is presented, i.e., how actual developers perceive the approach. In the future, it would be interesting to get some feedback of real software developers of how they perceive the approach and the tool, just as Bavota et al. [14] evaluate their approach.

VI. Conclusions

Move method refactoring is popular, and existing tools provide good automated tool support for the conduction of move method refactorings. However, it is difficult to manually identify which methods should be moved. Although a number of approaches and tools have been proposed to identify such methods based on source code metrics, change history, and textual information, none of them has ever exploited conducted move method refactorings that contain rich information about the motivations for further refactorings.

To this end, in this paper we propose a new way to identify methods that should be moved. Whenever a method is moved, the approach recommends to move methods from the same class that are highly similar and closely related to the moved method. The rational is that similar and closely related methods should be moved together. We evaluate the proposed approach on open-source applications by comparing the recommended move method opportunities against a list of move method refactorings that are discovered from the involved applications by third-party tools (i.e., *RefactoringCrawler* and *REF-FINDER*). Our evaluation results show that the approach is accurate in recommending methods to be moved (average precision 76%) and in recommending destinations for such methods (average precision 83%). Our evaluation results also show that for a substantial percentage (27%) of move method refactorings, the proposed approach succeeds in identifying more refactoring opportunities.

Acknowledgments

The work is funded by the National Natural Science Foundation of China (No. 61272169, 61472034), Program for New Century Excellent Talents in University (No. NCET-13-0041), and Beijing Higher Education Young Elite Teacher Project (No. YETP1183).

REFERENCES

[1] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126–139, 2004.

[2] W. F. Opdyke, "Refactoring object-oriented frameworks," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1992.

[3] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison Wesley Professional, 1999.

[4] G. Murphy, M. Kersten, and L. Findlater, "How are java software developers using the elipse IDE?" *Software, IEEE*, vol. 23, no. 4, pp. 76–83, July 2006.

[5] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Trransactions on Softeare Engineering*, vol. 38, no. 1, pp. 5–18, JANUARY/FEBRUARY 2012.

[6] H. Liu, X. Guo, and W. Shao, "Monitor-based instant software refactoring," *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1112–1126, 2013.

[7] W. G. Griswold and D. Notkin, "Automated assistance for program restructuring," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 2, no. 3, pp. 228–269, July 1993.

[8] F. Tip, A. Kiezun, and D. Baeumer, "Refactoring for generalization using type constraints," in *Proceedings of the Eighteenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'03)*, Anaheim, CA, October 2003, pp. 13–26.

[9] T. Mens, N. V. Eetvelde, and S. Demeyer, "Formalizing refactorings with graph transformations," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 17, no. 4, pp. 247–276, 2005.

[10] A. B. Binkley and S. R. Schach, "Validation of the coupling dependency metric as a predictor of run-time failures and maintenance measures," *Software Engineering, International Conference on*, vol. 0, p. 452, 1998.

[11] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Transactions on Software Engineering*, vol. 22, no. 10, pp. 751–761, 1996.

[12] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 347–367, 2009.

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

[14] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia, "Methodbook: Recommending move method refactorings via relational topic models," *Software Engineering, IEEE Transactions on*, vol. 40, no. 7, pp. 671–694, July 2014.

[15] F. Simon, F.Steinbrucker, and C.Lewerentz, "Metrics based refactoring," in *Proceedings of Europen Conference on Software Maintenance and Reengineering*, 2001, pp. 30–38.

[16] O. Seng, J. Stammel, and D. Burkhart, "Search-based determination of refactorings for improving the class structure of object-oriented systems," in *In Proceedings of the 8th annual conference on genetic and evolutionary computation*, 2006, pp. 1909–1916.

[17] V. Sales, R. Terra, L. Miranda, and M. Valente, "Recommending move method refactorings using dependency sets," in *Reverse Engineering (WCRE), 2013 20th Working Conference on*, Oct 2013, pp. 232–241.

[18] M. Zhang, T. Hall, and N. Baddoo, "Code bad smells: a review of current knowledge," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 23, no. 3, pp. 179–202, 2011. [Online]. Available: http://dx.doi.org/10.1002/smr.521

[19] H. Liu, Z. Niu, Z. Ma, and W. Shao, "Suffix tree-based approach to detecting duplications in sequence diagrams," *Software, IET*, vol. 5, no. 4, pp. 385–397, August 2011.

[20] N. Tsantalis and A. Chatzigeorgiou, "Identification of extract method refactoring opportunities for the decomposition of methods," *J. Syst. Softw.*, vol. 84, pp. 1757–1782, October 2011. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2011.05.016

[21] D. Silva, R. Terra, and M. T. Valente, "Recommending automated extract method refactorings," in *Proceedings of the 22Nd International Conference on Program Comprehension*, ser. ICPC 2014. New York, NY, USA: ACM, 2014, pp. 146–156. [Online]. Available: http://doi.acm.org/10.1145/2597008.2597141

[22] J. A. Dallal, "Identifying refactoring opportunities in object-oriented code: A systematic literature review," *Information and Software Technology*, no. 0, pp. –, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584914001918

[23] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *Proceedings of the 20th IEEE International Conference on Software Maintenance*, 2004, pp. 350–359.

[24] M. Munro, "Product metrics for automatic identification of "bad smell" design problems in java source-code," in *Software Metrics, 2005. 11th IEEE International Symposium*, Sept. 2005, pp. 15–15.

[25] N. Tsantalis and A. Chatzigeorgiou, "Identification of refactoring opportunities introducing polymorphism," *Journal of Systems and Software*, vol. 83, no. 3, pp. 391–404, 2010.

[26] G. Bavota, A. D. Lucia, and R. Oliveto, "Identifying extract class refactoring opportunities using structural and semantic cohesion measures," *Journal of Systems and Software*, vol. 84, no. 3, pp. 397 – 414, 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121210003195

[27] G. Bavota, R. Oliveto, A. De Lucia, G. Antoniol, and Y. Gueheneuc, "Playing with refactoring: Identifying extract class opportunities through game theory," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*, sept. 2010, pp. 1 –5.

[28] H. Liu, Z. Niu, Z. Ma, and W. Shao, "Identification of generalization refactoring opportunities," *Automated Software Engineering*, vol. 20, no. 1, pp. 81–110, 2013. [Online]. Available: http://dx.doi.org/10.1007/s10515-012-0100-0

[29] S. Demeyer, S. Ducasse, and O. Nierstrasz, "Finding refactorings via change metrics," in *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '00. New York, NY, USA: ACM, 2000, pp. 166–177.

[30] Z. Xing and E. Stroulia, "Umldiff: an algorithm for object-oriented design differencing," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ser. ASE '05. New York, NY, USA: ACM, 2005, pp. 54–65. [Online]. Available: http://doi.acm.org/10.1145/1101908.1101919

[31] Z. Xing and E. Stroulia,"Refactoring detection based on umldiff change-facts queries," in *13th Working Conference on Reverse Engineering*, Oct 2006, pp. 263–274.

[32] P. Weissgerber and S. Diehl, "Identifying refactorings from source-code changes," in *Automated Software Engineering, 2006. ASE '06. 21st IEEE/ACM International Conference on*, Sept 2006, pp. 231–240.

[33] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, "Automated detection of refactorings in evolving components," in *Proceedings of 20th European Conference on Object-Oriented Programming (ECOOP'06)*, ser. Lecture Notes in Computer Science, vol. 4067. Nantes, France: Springer Berlin / Heidelberg, July 3-7 2006, pp. 404–42.

[34] A. Broder, "On the resemblance and containment of documents," in *Proceedings of the Compression and Complexity of Sequences 1997*, ser. SEQUENCES '97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 21–29.

[35] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim, "Template-based reconstruction of complex refactorings," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*, Sept 2010, pp. 1–10.

[36] INTT: Identifier Name Tokenisation Tool. http://oro.open.ac.uk/id/eprint/28352.

[37] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Improving the tokenisation of identifier names," in *European Conference on Object-Oriented Programming (ECOOP 2011)*, ser. Lecture Notes in Computer Science, M. Mezini, Ed. Springer Berlin Heidelberg, 2011, vol. 6813, pp. 130–154. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-22655-7_7

[38] Hibernate. http://hibernate.org/.

[39] Weka. http://www.cs.waikato.ac.nz/ml/weka/.

[40] Apache Derby. http://db.apache.org/derby/.

[41] SQuirreL SQL Client. http://www.squirrelsql.org/.

[42] Phex. http://www.phex.org/mambo/.

[43] Camel. http://camel.apache.org/.

[44] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, "Automated detection of refactorings in evolving components," in *Proceedings of the 20th European Conference on Object-Oriented Programming*, ser. ECOOP'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 404–428. [Online]. Available: http://dx.doi.org/10.1007/11785477_24

[45] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim, "Template-based reconstruction of complex refactorings," in *Software Maintenance (ICSM), IEEE International Conference on*, Sept 2010, pp. 1–10.

[46] M. Godfrey and L. Zou, "Using origin analysis to detect merging and splitting of source code entities," *Software Engineering, IEEE Transactions on*, vol. 31, no. 2, pp. 166–181, Feb 2005.

[47] Q. D. Soetens, J. Pérez, S. Demeyer, and A. Zaidman, "Circumventing refactoring masking using fine-grained change recording," in *Proceedings of the 14th International Workshop on Principles of Software Evolution, IWPSE 2015, Bergamo, Italy, August 31 - September 4, 2015*, 2015, pp. 9–18.