Identifying Renaming Opportunities by Expanding Conducted Rename Refactorings

Hui Liu, Qiurong Liu, Yang Liu, Zhouding Wang

Abstract—To facilitate software refactoring, a number of approaches and tools have been proposed to suggest where refactorings should be conducted. However, identification of such refactoring opportunities is usually difficult because it often involves difficult semantic analysis and it is often influenced by many factors besides source code. For example, whether a software entity should be renamed depends on the meaning of its original name (natural language understanding), the semantics of the entity (source code semantics), experience and preference of developers, and culture of companies. As a result, it is difficult to identify renaming opportunities. To this end, in this paper we propose an approach to identify renaming opportunities by expanding conducted renamings. Once a rename refactoring is conducted manually or with tool support, the proposed approach recommends to rename closely related software entities whose names are similar to that of the renamed entity. The rationale is that if an engineer makes a mistake in naming a software entity it is likely for her to make the same mistake in naming similar and closely related software entities. The main advantage of the proposed approach is that it does not involve difficult semantic analysis of source code or complex natural language understanding. Another advantage of this approach is that it is less influenced by subjective factors, e.g., experience and preference of software engineers. The proposed approach has been evaluated on four open-source applications. Our evaluation results show that the proposed approach is accurate in recommending entities to be renamed (average precision 82%) and in recommending new names for such entities (average precision 93%). Evaluation results also suggest that a substantial percentage (varying from 20% to 23%) of rename refactorings are expansible.

Index Terms—Software Refactoring, Rename, Code Smells, Refactoring Opportunity, Identification.

1 INTRODUCTION

1.1 Software Refactoring

Software refactoring is a good means to improve software quality by restructuring the internal structure of software applications [1], [2]. Most of the modern IDEs, e.g., *Eclipse*, *IntelliJ IDEA* and *Visual Studio*, provide tool support for software refactoring. For example, *Eclipse* has a top-level menu item specially designed for software refactoring. It provides entries for dozens of software refactorings that are automated or semi-automated by *Eclipse*. Tool support is crucial for the success of software refactoring [3], [4], [5].

One of the key issues in software refactoring is to identify refactoring opportunities [1]. A refactoring opportunity is composed of three parts: source code that should be restructured, its restructuring plan (how to restructure the source code), and the motivation for such a refactoring (the cost and benefit). Only if refactoring opportunities have been identified, refactorings can be conducted.

However, manually identifying refactoring opportunities can be tedious and time-consuming, especially when such refactoring opportunities involve more than one file or package [1], [6]. Consequently, a number of algorithms and tools have been proposed to identify refactoring opportunities automatically or semi-automatically [7], [8], [1], [9].

1.2 Rename Refactoring and Tool Support

Among a dozen of well known refactorings, *rename* is the most popular one [10]. A survey conducted by Arnaoudova et al. [11] suggests that 39 percent of the developers involved in the survey apply rename refactorings from a few times per week to almost every day. According to the experiment results reported by Murphy et al. [10], all developers involved in their experiment have conducted rename refactorings whereas less than 60% of them have conducted the second popular refactoring *move* (moving software entities, such as methods, fields, and classes). One of the reasons why rename refactoring is so popular is that names of software entities are an important source for program comprehension [12]. According to the lexical analysis conducted by Deissenboeck and Pizka [13], one third of tokens in source code is identifiers, and such identifiers account for more than two thirds of the source code in terms of characters. The importance of high quality names in improving program readability and maintainability has been validated and acknowledged [14], [15].

As introduced in the preceding section, automated or semi-automated tools are needed to facilitate the identification of refactoring opportunities. The same is true for rename refactoring. However, automatically identifying

• *H. Liu, Q. Liu, Y. Liu, and Z. Wang are with the School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China*
E-mail: {liuhui08, liuqiurong}@bit.edu.cn, liuyangzj@126.com, zhoudingw@163.com

which entities should be renamed is not easy [6]. To determine whether a software entity should be renamed, automated approaches should determine 1) the meaning of the entity name; 2) the semantics of the entity; 3) experience and preference of the developer; 4) culture of the company where the software is developed. However, it is difficult to collect such information. First, it is challenging to understand the meaning of entity names because natural language understanding is difficult. Second, it is challenging to determine the semantics of software entities because semantic analysis of source code is difficult as well. Third, subjective factors like experience and preference of developers are difficult to collect, quantify or analyze. As a conclusion, it is difficult and inaccurate to identify renaming opportunities based on difficult semantic analysis of source code and difficult natural language understanding without considering subjective information of users.

Modern IDEs provide little support for identification of renaming opportunities. They provide strong automated tool support (refactoring engines) for conduction of rename refactorings [8]. However, these refactoring engines cannot work until renaming opportunities have been identified and refactoring solutions (new names of related entities) have been figured out.

To this end, in this paper we propose an approach to identify renaming opportunities by expanding conducted rename refactorings. Once a rename refactoring is conducted manually or with tool support, the proposed approach recommends to rename closely related software entities whose names are similar to that of the renamed entity. The rationale is that if an engineer makes a mistake in naming a software entity it is likely for her to make the same mistake in naming similar and closely related software entities.

The main advantage of the proposed approach is that it does not involve complex natural language understanding or difficult semantic analysis of source code. Another advantage is that it is less influenced by subjective factors, e.g., experience and preference of software engineers, because the identified renaming opportunities share the same context with the expanded rename refactoring.

This paper offers the following contributions:

- First, we propose a new way to identify renaming opportunities. To the best of our knowledge, it is the first one to identify refactoring opportunities by expanding a conducted refactoring.
- Second, the proposed approach has been implemented and validated on four open-source applications. Our evaluation results show that the proposed approach is accurate. 82% of recommended renaming opportunities have been actually performed later on; 93% of recommended solutions (i.e., new names) for such renaming opportunities were identical to those actually adopted later on. Our evaluation also shows that 21% of rename refactorings were successfully expanded by the proposed approach, and on average an expandable renaming led to two true renaming opportunities.

The rest of the paper is structured as follows. Section 2 illustrates why and how the proposed approach can identify renaming opportunities with a motivating example. Section 3 proposes the approach to identify renaming opportunities by expanding a conducted rename refactoring. Section 4 presents an evaluation of the proposed approach on open-source applications. Section 3.6 discusses related issues. Section 5 presents a short review of related research. Section 6 provides conclusions and potential future work.

2 Motivating Example

An example from the well-known open-source application *Weka*[1] is presented in Fig. 1. Source code of class $PreprocessPanel$ (version 3-3-4) is presented in Fig. 1 where irrelevant source code is omitted (notated as '//...').

According to the name of method *setBaseInstances* (on Line 6), we know that the main task of this method is to set local field *m_BaseInstances*. The first statement of this method *'m_BaseInstances = inst'* (Line 7 in Fig. 1) accomplishes this task. After that, the method initializes instances in different panels (*m_BaseInstPanel*, *m_AttPanel*, and *m_AttSummaryPanel*) with *m_BaseInstances* by calling method *setInstances* on these panels (Lines 9-11). Method *setInstances* looks as follows:

```
  public void setInstances(Instances inst) {
      m_Instances = inst;
     //...
}
```

The developer realized that fields initialized with the same value (the parameter *inst* of method *setBaseInstances*) are named differently in different classes: *'m_BaseInstances'* in class *PreprocessPanel* but *'m_Instances'* in other classes. Consequently, to make these names consistent [13], [16] she might rename the field *m_BaseInstances* of class *PreprocessPanel* to *'m_Instances'* by removing the term *'Base'* from it. We have discovered this rename refactoring by checking the evolution history of the class.

However, renaming the field alone was insufficient. First of all, the name of a *set method* had better be consistent with the name of the field it accesses. Consequently, once the field *m_BaseInstances* is renamed to *'m_Instances'*, the corresponding set method *setBaseInstances* had better be renamed to *'setInstances'*.

Second, a set of related set methods on Lines 14-43, e.g., *setBaseInstancesFromURL*, had better be renamed as well. The main tasks of these methods are to set the field *m_BaseInstances*. They set this field by calling *setBaseInstances*. Consequently, when *setBaseInstances* is renamed to *'setInstances'* as discussed in the preceding paragraph, these methods had better be renamed from

1. http://www.cs.waikato.ac.nz/ml/weka/

```
/* version 3-3-4 */
package weka.gui.explorer;
public class PreprocessPanel extends JPanel {
    protected Instances m_BaseInstances;
       //...
    public void setBaseInstances(Instances inst) {
      m_BaseInstances = inst;
          // ...
      m_BaseInstPanel.setInstances(m_BaseInstances);
      m_AttPanel.setInstances(m_BaseInstances);
      m_AttSummaryPanel.setInstances(m_BaseInstances);
          // ...
    }
    public void setBaseInstancesFromFileQ() {
          // ...
      setBaseInstancesFromFile(selected);
          // ...
    }
    public void setBaseInstancesFromDBQ() {
          // ...
      setBaseInstancesFromDB(InstQ);
          // ...
    }
    public void setBaseInstancesFromURLQ() {
          // ...
      setBaseInstancesFromURL(url);
          // ...
    }
    public void  setBaseInstancesFromFile(final File f)  {
          // ...
      setBaseInstances(new Instances(r));
          // ...
    }
    public void  setBaseInstancesFromDB(final InstanceQuery iq)  {
          // ...
      setBaseInstances(new Instances(i));
          // ...
    }
    public void setBaseInstancesFromURL(final URL u) {
          // ...
      setBaseInstances(new Instances(r));
          // ...
    }
       //...
}
```

Fig. 1. Motivating Example from Open-Source Application *Weka*

*setBaseInstancesFrom** to *'setInstancesFrom*'* (removing the term *'Base'*) to keep them consistent with the field it accesses and sibling methods they call.

As a result, she should conduct eight rename refactorings (all of them have been discovered in *Weka* 3-3-5), and all of these refactorings are caused by the same reason: an inappropriately named field. However, when the first rename refactoring is conducted, existing refactoring tools cannot recommend other renaming opportunities. Consequently, the developer has to manually identify all of such fields and methods to rename. However, manual identification is tedious and time-consuming because the class contains more than 900 lines of source code and the involved software entities are declared in different locations. As a result, manual identification might miss some renaming opportunities and thus result in incomplete renaming.

The proposed approach analyzes the first rename refactoring, i.e., renaming field *m_BaseInstances* to *'m_Instances'*, and learns that the refactoring is to remove term 'Base'. It collects closely related software entities (methods that access this field) whose name contains the deleted term 'Base' and its sibling term 'Instances'. Among all such entities, it recommends method *setBaseInstances* to be renamed because its name is more similar to *'m_BaseInstances'* than names of other entities. When the recommended renaming opportunity is accepted, the proposed approach continues to recommend until its recommendation is rejected or no entity could be recommended. For the example presented in Fig. 1, all of the seven renaming opportunities can be recommended successfully in this way.

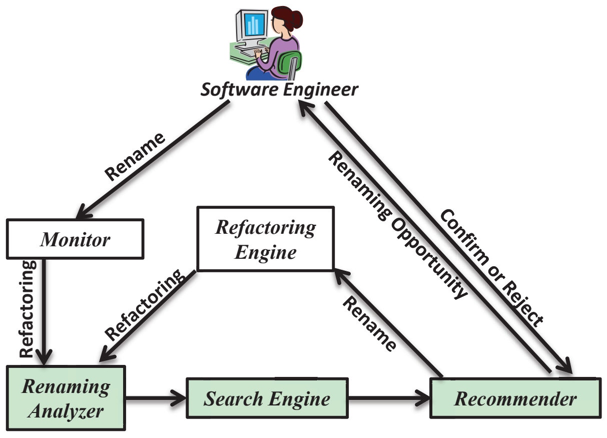

Fig. 2. Overview of the Approach

Details on how the proposed approach works for the motivating example are presented in Section 3 where this example is used to illustrate how the proposed approach recommends renaming opportunities, and how new names are generated.

3 Approach

3.1 Overview

An overview of the proposed approach is presented in Fig. 2. As suggested by the figure, the proposed approach works as follows:

1) The software engineer applies a rename refactoring r_1 on software entity e, and changes its name from $oldName$ to $newName$. She might do it manually or semi-automatically with refactoring tools.
2) A monitor running in the background captures the rename refactoring r_1.
3) A renaming analyzer analyzes r_1, and generates an edit script that transforms $oldName$ into $newName$.
4) A search engine searches for software entities that are closely related to e. If names of such entities are similar to $oldName$, such entities are considered as potential renaming opportunities.
5) Among all potential renaming opportunities, the recommender selects the one whose name is the most similar to $oldName$.
6) The software engineer manually checks the recommended renaming opportunity. If it is rejected, recommendation triggered by r_1 stops. Otherwise, a refactoring engine (which is usually embedded in IDEs) conducts the corresponding rename refactoring r_2, and new renaming opportunities would be recommended based on both r_1 and r_2. The recommendation continues until a recommended opportunity is rejected or no opportunities can be recommended. A rejected opportunity stops the recommendation because it suggests that the approach might not work for the given refactoring.

Refactoring engines are distributed with modern IDEs. Monitors are also available [6], [17] and conducted rename refactorings could be identified with existing algorithms [18], [11], [19], [20], [21] that are introduced in Section 5.3. Consequently, the proposed approach focuses on renaming analyzer, search engine, and recommender only. Details of these parts are introduced in the following sections.

3.2 Analysis of Rename Refactorings

A rename refactoring that changes the name of software entity e from $oldName$ to $newName$ is represented as:

$$r_1 =< e, oldName, newName > \tag{1}$$

$oldName$ is composed of a sequence of terms:

$$oldName =< a_1, a_2, \ldots, a_n > \tag{2}$$

$oldName$ is decomposed into a sequence of terms delimited by underscores and capital letters assuming that the name follows the popular camel case or snake case naming convention. Alternative approaches to decompose identifier names that do not follows these conventions are discussed in Section 3.6.1.

By comparing $oldName$ against $newName$, we can get the smallest set of deletions, insertions, and replacements that transforms $oldName$ into $newName$.

A deletion command is represented as

$$d_{(i,k)} =< i, k > \tag{3}$$

suggesting that the sequence of terms $a_i \ldots a_k$ is removed from $oldName$.

A replacement command is represented as

$$rp_{(i,k)} =< i, k, b_j, b_{j+1}, \ldots, b_{j+t} > \tag{4}$$

suggesting $a_i \ldots a_k$ is replaced with a sequence of terms $b_j b_{j+1} \ldots, b_{j+t}$.

An insertion command is represented as

$$sc_i =< i, b_j, b_{j+1}, \ldots, b_{j+q} > \tag{5}$$

suggesting a sequence of terms $b_j b_{j+1} \ldots, b_{j+t}$ is inserted immediately after a_i. If the sequence is inserted at the beginning of a name, $i = 0$.

For the motivating example presented in Section 2, the captured rename refactoring is to rename field $m\_BaseInstances$ to $'m\_Instances'$. Consequently, the only edition command is to delete the term 'Base' (the third term of $m\_BaseInstances =$ $<' m',' \_',' Base',' Instances' >$):

$$d_{(3,3)} =< 3, 3 > \tag{6}$$

Tokenisation of identifier names usually takes '_' as a separator character and thus removes such characters in resulting tokens (terms) [22]. However, it is possible that software engineers rename a software entity by adding or removing '_' only. Consequently, the proposed approach keeps these characters to capture such small changes.

3.3 Searching for Related Entities

Entities that are related to e in at least one of the following ways are considered as closely related entities:

- Inclusion
  Entities that are directly included by e and elements that directly include e. For example, if e is a class, all methods and fields within this class are considered as closely related entities.
- Sibling
  Siblings of e. For example, if e is a method, all methods and fields within the same class are considered as closely related entities.
- Reference
  All entities that are referred by e and entities that refer to e. For example, if e is a method, all methods that are invoked within this method and all methods that invoke e are taken as closely related entities.
- Inheritance
  If e is a class, its superclass and subclasses are taken as closely related entities.

All such closely related entities are collected into a set that is notated as res (<u>r</u>elated <u>e</u>ntitie<u>s</u>).

For the motivating example presented in Section 2, the renamed entity is the field $m\_BaseInstances$. Its closely related entities include:

- Its enclosing class: *PreprocessPanel*;
- Its siblings: all fields and methods of class *PreprocessPanel* (except for $m\_BaseInstances$ itself);
- Methods that directly access this field: methods *setBaseInstances*, *getFilters*, and *setWorkingInstancesFromFilters* of class *PreprocessPanel*.

3.4 Recommendation

3.4.1 Preconditions

An element (software entity) e' in res whose name is $cName$ is considered for recommendation only if it meets all of the following conditions:

- For every deletion command $d_{(i,k)}$ that is derived from the renaming analysis in Section 3.2, $cName$ should contain either $a_{i-1}\cdots a_k$ or $a_i\cdots a_{k+1}$. In other words, $cName$ should contain the sequence of deleted terms ($a_i\cdots a_k$) and at least one of its sibling terms (context).
- For every replacement command $rp_{(i,k)}$ derived from the renaming analysis in Section 3.2, $cName$ should contain the sequence of replaced terms ($a_i\cdots a_k$) and one of its sibling terms. Consequently, $cName$ must contain either $a_{i-1}\cdots a_k$ or $a_i\cdots a_{k+1}$.
- For every insertion command sc_i derived from the renaming analysis in Section 3.2, $cName$ should contain a_ia_{i+1}. In other words, $cName$ should contain both of the direct neighbors of the inserted terms.

Entities that fail to meet any of the preconditions listed above are removed from res.

The name of entity e' should contain both deleted (or replaced) terms and their sibling terms because such sibling terms can help to improve accuracy of the recommendation. According to our software development experience, a deletion or replacement command in rename refactorings often removes or replaces a single term. However, this single term is usually not enough to capture the refactoring context. Consequently, we concatenate this term with at least one of its sibling terms to represent the refactoring context. An insertion command does not replace or remove any terms. Consequently, we use both a_i and a_{i+1} to present the insertion point for insertion command sc_i.

For the motivating example presented in Section 2, the only edition command is $d_{(3,3)}$. For this command, a_j is the term 'Base', $a_{j-1}a_j =' \_Base'$, and $a_ja_{j+1} = 'BaseInstance'$. Among all the related entities found in Section 3.3, the following entities meet all preconditions that are listed and discussed in preceding paragraphs:

- Method *setBaseInstances* of class *PreprocessPanel;*
- Method *setBaseInstancesFromFileQ* of class *PreprocessPanel;*
- Method *setBaseInstancesFromDBQ* of class *PreprocessPanel;*
- Method *setBaseInstancesFromURLQ* of class *PreprocessPanel;*
- Method *setBaseInstancesFromFile* of class *PreprocessPanel;*
- Method *setBaseInstancesFromDB* of class *PreprocessPanel;*
- Method *setBaseInstancesFromURL* of class *PreprocessPanel;*

Other related software entities that fail to meet the preconditions are excluded for further consideration.

3.4.2 Segmentation

The original name ($oldName = a_1\cdots a_n$) of the renamed entity e and the name ($cName = c_1\cdots c_m$) of entity e' from res are segmented into subsequences to facilitate the computation of their similarity.

For every deletion command $d_{(i,k)}$ that is derived from the renaming analysis in Section 3.2, k is a segmentation point for $oldName$ suggesting that the string should be segmented at the position of k. Its corresponding segmentation point for $cName$ is p where $c_{p+i-k-1}\cdots c_p$ is identical to $a_{i-1}\cdots a_k$ or $c_{p+i-k}\cdots c_{p+1}$ is identical to $a_i\cdots a_{k+1}$. These two segmentation points together with the length of the deleted terms are called a segmentation pair $< k, p, k-i+1 >$.

For every replacement command $rp_{(i,k)}$ that is derived from the renaming analysis in Section 3.2, k is a segmentation point for $oldName$. Its corresponding segmentation point for $cName$ is p where $c_{p+i-k-1}\cdots c_p$ is identical to $a_{i-1}\cdots a_k$ or $c_{p+i-k}\cdots c_{p+1}$ is identical to $a_i\cdots a_{k+1}$. These two segmentation points together with the length of the replaced terms are called a segmentation pair $< k, p, k-i+1 >$.

For every insertion command $sc_{(i)}$ that is derived from the renaming analysis in Section 3.2, i is a segmentation point for $oldName$. Its corresponding segmentation point for $cName$ is p where c_pc_{p+1} is identical to a_ia_{i+1}. Its corresponding segmentation pair is $< i, p, 0 >$ where 0 suggests that no term has been deleted by the command.

Segmentation pairs for all of the commands are represented as a sequence:

$$sps =< sp_1, sp_2, \ldots, sp_v > \tag{7}$$
$$sp_i =< p_i, q_i, len_i > \tag{8}$$
$$p_i < p_{i+1} \tag{9}$$

Where v is the number of edition commands and sp_i is the ith segmentation pair. In *sps*, all segmentation pairs are sorted in ascending order according to their first elements (p_i).

These segmentation pairs segment $oldName$ and $cName$ into $v + 1$ subsequences, respectively:

$$sg_i(oldName) = a_{(p_{i-1}+1)} \cdots a_{p_i} \tag{10}$$
$$sg_i(cName) = c_{(q_{i-1}+1)} \cdots c_{q_i} \tag{11}$$

where $sg_i(oldName)$ is the ith subsequence of $oldName$, $sg_i(cName)$ is the ith subsequence of $cName$, and $p_0 = q_0 = 0$.

For an edition command, there might exist more than one segmentation pair. For example, for deletion command $d_{(i,k)}$, if $a_{i-1} \cdots a_k$ appears twice in $cName$, there are two segmentation pairs for this command. However, for a single command, only one segmentation pair can be inserted into a segmentation pair sequence (*sps*). Since selecting different segmentation pairs for the command may result in different similarity between $oldName$ and $cName$ as computed in the next section (Section 3.4.3), we select the one that maximizes the similarity.

For the motivating example presented in Section 2, we take the method *setBaseInstances* to illustrate how identifier names are segmented. In this example, the only edition command is $d(3, 3)$. Consequently, both $oldName$ and $cName$ should be segmented into two subsequences:

$$oldName =<' m',' \_',' Base',' Instances' > \tag{12}$$
$$cName =<' set',' Base',' Instances' > \tag{13}$$
$$sps =< sp_1 > \tag{14}$$
$$sp_1 =< 3, 2, 1 > \tag{15}$$
$$sg_1(oldName) =' m\_Base' \tag{16}$$
$$sg_2(oldName) =' Instances' \tag{17}$$
$$sg_1(cName) =' setBase' \tag{18}$$
$$sg_2(cName) =' Instances' \tag{19}$$

3.4.3 Similarity and Recommendation

Similarity between $oldName =< a_1, a_2, \ldots, a_n >$ and $cName =< c_1, c_2, \ldots, c_m >$is computed as follows:

$$sim(oldName, cName) = \frac{1}{v+1} \times \sum_{i \in [1,v+1]} S(sg_i(oldName), sg_i(cName)) \tag{20}$$

where $sg_i(oldName)$ is the ith subsequence of $oldName$, $sg_i(cName)$ is the ith subsequence of $cName$, and v is the number of edition commands (the same as that in Equation 7). $S(sg_i(oldName), sg_i(cName))$ is the text similarity between two sequences of terms:

$$S(seq_1, seq_2) = \frac{2 \times |seq_1 \cap seq_2|}{|seq_1| + |seq_2|} \tag{21}$$

$|seq_1 \cap seq_2|$ is the number of terms appearing in both seq_1 and seq_2.

Among all entities in res, the one e' with the greatest similarity with $oldName$ is recommended.

For entity (method *setBaseInstances*) in the motivating example presented in Section 2, the similarity is computed as follows:

$$oldName = <' m',' \_',' Base',' Instances' > \tag{22}$$
$$cName = <' set',' Base',' Instances' > \tag{23}$$

$$\begin{aligned} sim(oldName,cName) = & \\ & \frac{1}{2} \times [S(sg_1(oldName), sg_1(cName) \\ & + S(sg_2(oldName), sg_2(cName))] \\ = & \frac{1}{2} \times [S('m\_Base',' setBase') \\ & + S('Instances',' Instances')] \\ = & \frac{1}{2} \times [\frac{2 \times 1}{3+2} + \frac{2 \times 1}{1+1}] \\ = & 0.7 \end{aligned} \tag{24}$$

3.4.4 Generation of New Names

To generate the new name for recommended entity e', we generate a set of edition commands to transform its original name $cName$ to its recommended new name. We get such edition commands by adjusting positions of edition commands that have transformed $oldName$ to $newName$:

- For each deletion command $d_{(i,k)}$ that is derived from the renaming analysis in Section 3.2, we generate a deletion command $cd_{(p-k+i,p)}$ where $< k, p, k - i >$ is the corresponding segmentation pair defined in Section 3.4.2.
- For each replacement command $rp_{(i,k)} = < i, k, b_j, b_{j+1}, \ldots, b_{j+t} >$ that is derived from the renaming analysis in Section 3.2, we generate a replacement command $crp_{(p-k+i,p)} = < p-k+i, p, b_j, b_{j+1}, \ldots, b_{j+t} >$ where $< k, p, k-i >$ is the corresponding segmentation pair.
- For each insertion command $sc_i = < i, b_j, b_{j+1}, \ldots, b_{j+q} >$ that is derived from the renaming analysis in Section 3.2, we generate an insertion command $csc_p =< p, b_j, b_{j+1}, \ldots, b_{j+q} >$ where $< i, p, 0 >$ is the corresponding segmentation pair.

After all such edit commands are conducted on $cName$, the resulting string is the recommended new name for the entity e'.

For the motivating example presented in Section 2, method *setBaseInstances* is recommended for renaming. In this example, there is only one edition command $d_{(3,3)}$ that has been conducted on $oldName$. For this deletion command d_3, its corresponding segmentation pair is $< 3, 2, 1 >$. Consequently, the deletion command on $cName$ should be $d_{(2,2)}$, i.e., removing the second term from $cName$. Applying such command on $cName$ results in the recommended new name: *setInstances*. The recommended name is identical to that appearing in Weka 3-3-5, suggesting that the recommendation is correct.

3.5 Further Recommendation

In Sections 3.2-3.4, the proposed approach recommends a renaming opportunity based on a conducted rename refactoring r_1. If the initial recommendation is accepted, and software engineers carry out the recommended rename refactoring r_2, the proposed approach tries to recommend new renaming 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 renaming opportunity or a recommended opportunity is rejected.

Suppose that based on refactoring r_1, a sequence of refactorings $acceptR =< r_2, r_3, \cdots, r_k >$ has been recommended and conducted. The proposed approach searches for related entities as follows:

$$res = res(r_1) \cup res(r_2) \cdots \cup res(r_k) \tag{25}$$

where $res(r_i)$ is a set of software entities collected according to refactoring r_i. The collection of $res(r_i)$ is the same as that introduced in Section 3.3.

For every entity e'' from res, the similarity between its name $eName$ and the original names involved in $r_1, r_2, r_3, \cdots, r_k$ is calculated as follows:

$$sim(eName) = \frac{1}{k} \times \sum_{i \in [1,k]} sim(name_i, eName) \tag{26}$$

where $name_i$ is the original name of the entity renamed by refactoring r_i, and $sim(name_i, eName)$ is the text similarity between two identifier names that is defined in Equation 20.

Among all entities in res, the one with the greatest similarity is recommended. Its new name is generated based on r_1 in the same way as introduced in Section 3.4.4.

For the motivating example presented in Section 2, once the recommended rename refactoring on method *setBaseInstances* has been conducted, the proposed approach should try to recommend new renaming opportunities according to rename refactorings on both *setBaseInstances* and *m_BaseInstances*. We take method *setBaseInstancesFromFile* as an example to illustrate how to compute its similarity with more than one conducted rename refactorings:

$$eName =' setBaseInstancesFromFile' \tag{27}$$
$$name_1 =' m\_BaseInstances' \tag{28}$$
$$name_2 =' setBaseInstances' \tag{29}$$
$$sim(name_1, eName) = \frac{1}{2} \times (\frac{2 \times 1}{5} + \frac{2 \times 1}{4}) = 0.45 \tag{30}$$
$$sim(name_2, eName) = \frac{1}{2} \times (\frac{2 \times 2}{4} + \frac{2 \times 1}{4}) = 0.75 \tag{31}$$
$$\begin{aligned} sim(eName) &= \frac{1}{2} \times \sum_{i \in [1,2]} sim(name_i, eName) \\ &= \frac{1}{2} \times (0.45 + 0.75) = 0.6 \end{aligned} \tag{32}$$

3.6 Limitations

3.6.1 *Decomposition of Identifier Names*

The proposed approach assumes that identifier names follow the popular camel case or snake case naming convention. As a result, it decomposes such names according to capital letters and underscores. 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*[2] proposed by Butler et al.[22] can decompose identifier names into meaningful terms even if these names do not follow camel case naming convention.

3.6.2 *Synonym Database*

The proposed approach computes text similarity between two strings by counting the common terms appearing in both strings. However, it does not take synonyms into consideration. For example, comparing strings 'get' and 'retrieve', the proposed approach generates the similarity of zero although these terms are synonyms.

On one side, synonym databases, e.g., *WordNet*[3], might help to improve the computation of text similarity. On the other side, synonym analysis might complicate the proposed approach and make it hard to understand or implement. The reason why synonym analysis in source code is complex is that terms 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 Evaluation

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

2. http://oro.open.ac.uk/28352/
3. http://wordnet.princeton.edu/

4.1 Research Questions

The evaluation investigates the following research questions:

- **RQ1**: How often are renaming opportunities recommended by the proposed approach accepted?
- **RQ2**: How often are new names generated by the proposed approach accepted?
- **RQ3**: How often can a rename refactoring be expanded to discover more renaming opportunities?

RQ1 and **RQ2** concern the precision of the proposed approach in recommending renaming opportunities and in recommending new names for entities to be renamed. These research questions are important because the proposed approach would be useless if it overwhelms developers with a large number of false positives (resulting from low precision). **RQ3** concerns how often developers need the proposed approach if it is accurate. Only if a non-trivial percentage of rename refactorings can be expanded, the proposed approach can be used frequently and thus create value. Otherwise, the approach might be rarely used. As a conclusion, investigating these questions would reveal whether the proposed approach is really useful.

4.2 Subject Applications

An overview of the subject applications is presented in Table 1.

Weka[4] 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. 70 versions of this application (from 3.0 to 3.7.11) have been involved in the evaluation. The size of the application varies from 38,890 to 272,212 LOC. Source code of this application was downloaded from its SVN server (https://svn.cms.waikato.ac.nz/svn/weka).

Hibernate[5] is an open-source application developed by Red Hat[6]. The purpose of this project is to provide an easy way to achieve persistence in Java. 113 versions of this application (from v30alpha to 4.3.6.Final) have been involved in the evaluation. The size of the application varies from 51,276 to 200,874 LOC. Source code of this application was downloaded from SourceForge (http://sourceforge.net/projects/hibernate).

Derby[7] is an Apache DB subproject sponsored by Apache Software Foundation[8]. It is an open-source relational database implemented entirely in Java. 23 versions of this application (from 10.0.2.1 to 10.10.2.0) have been involved in the evaluation. The size of the application varies from 258,295 to 626,560 LOC. Source code of this application was downloaded from its SVN server (http://svn.apache.org/repos/asf/db/derby/).

Camel[9] is an open-source integration framework based on enterprise integration patterns. The application was implemented in Java, and 34 versions of this application (from 1.0.0 to 2.11.0) have been involved in the evaluation. The application is large and complex. Consequently, in this evaluation we only analyzed the basic module *Camel-core*. The size of the module varies from 9,125 to 84,041 LOC. Source code of this application was downloaded from its SVN server (http://svn.apache.org/repos/asf/camel/tags/).

```
For each application $app_j$
    Collect rename refactorings $(RS)$ from $app_j$
    For each renaming $rs_i$ in $RS$
        $rhs = < rs_i >$
        While true
            Based on $rhs$ the approach recommends
            renaming $rn = < e, oldName, newName >$
            If $rn = null$
                $break$
            End of If
            If $e$ has been removed
                $rn$ is inconclusive
                $continue$
            End of If
            If exist $r = < e, oldName, xName >$ and $r \in RS$
                Renaming opportunity $e$ is accepted and
                $r$ is moved from $RS$ to t $rhs$
                If $xName$=$newName$
                    Recommended name $newName$ is accepted
                else
                    $newName$ is rejected
                End of If
            else
                Renaming opportunity $e$ is rejected
                $break$
            End of If
        End of While
        Remove $rhs$ from $RS$
    End of For
End of For
```

Fig. 3. Process of the Evaluation

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 rename refactorings have been conducted on these applications.

4.3 Process

Rename refactorings (notated as *RS*) that had been conducted on subject applications were discovered by three participants with semi-automated tool support. The three participants were master level students who majored in computer science. They were familiar with software refactoring, and all of them had participated in

4. http://www.cs.waikato.ac.nz/ml/weka/
5. http://hibernate.org/
6. http://www.redhat.com/en
7. http://db.apache.org/derby/
8. http://www.apache.org/
9. http://camel.apache.org/

TABLE 1
Subject Applications

Subject Application	Domain	No. of Versions	Size (varies from version to version)
Weka	Machine Learning	70	38,890 ∼ 272,212 LOC
Hibernate	Persistence	113	51,276 ∼ 200,874 LOC
Derby	Database	23	258,295 ∼ 626,560 LOC
Camel	Integration framework	34	9,125 ∼ 84,041 LOC

at least one refactoring-related project before the evaluation. Tools used by them to discover rename refactorings include *RefactoringCrawler* [20], *REF-FINDER* [23] and *Eclipse*, all of which are publicly available. Although tools that were specially designed to detect rename refactorings, i.e., *renaming detector* [18] and *REPENT* [11], have been introduced, they are not publicly available. Consequently, general-purpose refactoring detectors, i.e., *RefactoringCrawler* and *REF-FINDER*, were used instead in the evaluation. More information about existing approaches and tools that can be used to identify rename refactorings is presented in Section 5.1.

For each of the potential rename refactorings identified by *RefactoringCrawler* or *REF-FINDER*, all of the three participants manually checked related files together. By comparing related entities in different versions, they were asked to decide whether it was a real rename refactoring or not. For example, if *RefactoringCrawler* suggested that method m_1 in version v_1 was renamed to m_2 in version v_2, they were asked to compare the two methods m_1 and m_2, and decide whether m_2 should be viewed as a renamed version of m_1 (instead of a newly introduced method), which is well known as origin analysis [24]. They were also asked to decide whether multiple closely related renaming refactorings should be combined as a single refactoring. For example, if a method was renamed, all of its overriding and overridden methods must be renamed in the same way. Although *RefactoringCrawler* and *REF-FINDER* would report a potential rename refactoring for each of the renamed methods, they had better be taken as a single rename refactoring. The same is true for the renaming of classes and their constructors. In case of diverging decisions on a potential rename refactoring, the three participants were requested to discuss (or vote if needed).

The process of the automated evaluation based on the discovered rename refactorings is presented in Fig. 3. First, the proposed approach recommended renaming opportunities by expanding each rename refactoring in RS. If the approach recommended software entity e to be renamed, and there is a rename refactoring r in RS that renamed e (to $xName$), the recommended renaming opportunity was accepted. If it recommended software entity e to be renamed but e was removed from the next version, the recommendation was inconclusive and thus it was ignored. If it recommended software entity e and the name of this entity did not change, the recommended renaming opportunity was rejected.

If a renaming opportunity was accepted, the recommended new name was accepted if and only if this name was identical to $xName$. In other words, a recommended new name was accepted if and only if the recommended rename refactoring was identical to one of the refactorings that had been actually conducted on the application.

The recommendation triggered by rs_i continued as stated in Section 3.5 if the previous recommendation was accepted or ignored. It stopped when its previous recommendation was rejected or no recommendation could be generated.

4.4 Measurements

To answer research question **RQ1**, we calculated the precision of the proposed approach in recommending renaming opportunities as follows:

$$P_{opp} = \frac{number\ of\ accepted\ opportunities}{number\ of\ recommended\ opportunities} \tag{33}$$

To answer research question **RQ2**, we calculated the precision of the proposed approach in recommending new names for accepted renaming opportunities as follows:

$$P_{name} = \frac{number\ of\ accepted\ names}{number\ of\ accepted\ opportunities} \tag{34}$$

To answer research question **RQ3**, we measured what percentage of rename refactorings can be expanded:

$$R_{exp} = \frac{N_{expanded}}{N_{tried}} \tag{35}$$

Where N_{tried} is the number of rename refactorings that the proposed approach tried to expand, and $N_{expanded}$ is the number of refactorings that the proposed approach expanded successfully. A special case is that based on a rename refactoring (r_1), the proposed approach successfully recommended more than one rename refactoring (e.g., $r_2, r_3, \ldots, r_k$). In this case, r_1 is a tried and expanded refactoring (i.e., it was counted in both N_{tried} and $N_{expanded}$) whereas the recommended refactorings ($r_2, r_3, \ldots, r_k$) were neither tried refactorings nor expanded ones (i.e., they were not counted in N_{tried} or $N_{expanded}$). As a result, N_{tried} is always smaller than the number of rename refactorings discovered in subject applications (N_1).

Whenever a software entity is renamed, the proposed approach recommends to rename closely related entities whose identifier names are similar or identical to that of the renamed entity. We define the following metrics

TABLE 2
Evaluation Results

	Hibernate	*Weka*	*Derby*	*Camel*	Total
Discovered Rename Refactorings (N_1)	105	74	43	73	295
Recommended Opportunities (N_2)	29	31	13	32	105
Accepted Opportunities (N_3)	23	26	11	26	86
Rejected Opportunities	6	4	1	6	17
Inconclusive Opportunities	0	1	1	0	2
$P_{opp} = N_3/N_2$	79%	84%	85%	81%	82%
Accepted New Names (N_4)	20	25	9	26	80
$P_{name} = N_4/N_3$	87%	96%	82%	100%	93%
Accepted Out-of-File Recommendation (N_5)	5	0	3	3	11
$R_{out} = N_5/N_3$	22%	0%	27%	12%	13%
Expanded Refactorings (N_6)	16	10	7	10	43
Tried Refactorings (N_7)	82	47	31	47	207
$R_{exp} = N_6/N_7$	20%	21%	23%	21%	21%
$R_{identical}$	9%	0%	27%	27%	14%

to measure how often the recommended entity and the originally renamed entity have the same identifier name:

$$R_{identical} = \frac{N_{identical}}{number\ of\ accepted\ rename\ refactorings} \quad (36)$$

where $N_{identical}$ is the number of recommended (and then accepted) entities that have the exact same name as the originally renamed entity.

We also counted the number (N_{out}) of accepted renaming opportunities that appeared out of the file where their corresponding expanded renaming refactorings were conducted. We call such renaming opportunities *accepted out-of-file recommendation*. In other words, if an accepted renaming opportunity r_{opp} was recommended based on conducted refactoring r_1 and they distributed in different files, r_{opp} is counted in N_{out}. We also calculated the ratio of N_{out} to the total number of accepted opportunities:

$$R_{out} = \frac{N_{out}}{number\ of\ accepted\ rename\ refactorings} \quad (37)$$

4.5 Results and Analysis

Evaluation results are presented in Table 2. From the table, we made the following observations:

- The proposed approach was accurate in recommending renaming opportunities. Its precision (P_{opp}) was 79%, 84%, 85%, and 81% on *Hiberate*, *Weka*, *Derby*, and *Camel*, respectively. On average 82% of the recommended renaming opportunities were accepted (answering research question *RQ1*).
- The proposed approach was accurate in recommending new names for entities to be renamed. The precision (P_{name}) was 87%, 96%, 82%, and 100% on *Hiberate*, *Weka*, *Derby*, and *Camel*, respectively. On average 93% of the new names generated by the proposed approach were accepted (answering research question *RQ2*).
- A substantial number of rename refactorings could be expanded to discover more renaming opportunities. On the subject applications, the proposed approach has tried to expand 207 rename refactorings whereas 43 of them have been expanded successfully (resulting in 86 true renaming opportunities). In other words, on average 21% of the renaming refactorings could be used to discover more renaming opportunities. On average, an expandable renaming would lead to 2(=86/43) true renaming opportunities. In other words, if you identify (and apply) ten renaming opportunities, the proposed approach would identify $4.2(= 10 \times 21\% \times 2)$ true renaming opportunities on average (answering research question *RQ3*).
- In most cases (86%=1-$R_{identical}$) the identifier name of the recommended entity is similar but not identical to that of the originally renamed entity. An exceptional case is the renaming of fields with *set* methods. It is likely that a field and the parameter of its *set* method have the same name. Once the field is renamed, the proposed approach would suggest to rename the parameter (and the *set* method) accordingly. Another exceptional case is the renaming of overloaded methods that have the same name but different parameters. Whenever an overloaded method is renamed, the proposed approach would suggest to rename other overloaded methods with the same name. The third exceptional case is that some methods are named with noun phrases and share the same name with fields accessed by such methods. In the evaluation, among the cases where the identifier name of the recommended entity is identical to that of the originally renamed entity, these three exceptional cases accounted for 42%, 25%, and 33%, respectively.
- Around 13% of the accepted renaming opportunities appeared out of the file where their corresponding expanded renaming refactorings were conducted. In other words, in most cases (87%) such opportunities were recommended based on renamings within the same file. The results suggest that it could be efficient and effective to search renaming opportunities within the document where the conducted renaming occurs before extending the search scope to the whole application.

We also manually checked the rejected renaming recommendations, and our analysis results suggest that around a quarter (4/17) of the rejected recommendations are valuable because they identify some entities that should have been renamed. As stated in Section 4.3, in the evaluation if the approach recommended a software entity to be renamed but its name did not change in the following versions, the recommended renaming opportunity was taken as a *rejected* one. However, the fact

```
 1  /* version 2.2.0  */
 2  public class RedeliveryPolicy implements Cloneable, Serializable  {
 3          //...
 4     protected long redeliverDelay = 1000L;
 5          //...
 6     public void setRedeliverDelay(long redeliverDelay) {
 7          this.redeliverDelay = redeliverDelay;
 8          // if max enabled then also set max to this value in case max was too low
 9          if (maximumRedeliveryDelay > 0 && redeliverDelay > maximumRedeliveryDelay) {
10              this.maximumRedeliveryDelay = redeliverDelay;
11          }
12      }
13
14          //...
15  }
16
17  /* version 2.3.0  */
18  public class RedeliveryPolicy implements Cloneable, Serializable {
19          //...
20     protected long redeliveryDelay = 1000L;
21          //...
22     public void setRedeliveryDelay(long redeliverDelay) {
23          this.redeliveryDelay = redeliverDelay;
24          // if max enabled then also set max to this value in case max was too low
25          if (maximumRedeliveryDelay > 0 && redeliverDelay > maximumRedeliveryDelay) {
26              this.maximumRedeliveryDelay = redeliverDelay;
27          }
28      }
29
30          //...
31  }
```

Fig. 4. Incomplete Renaming in *Camel*

that an entity has not been renamed does not necessarily suggest that it should not be renamed.

An example is presented in Fig.4. At first, the field *redeliverDelay* (Line 4) of class *org.apache.camel.processor.RedeliveryPolicy* was renamed to *redeliveryDelay* (Line 20), i.e., the verb 'redeliver' was replaced with a noun 'redelivery' to modify the noun 'Delay'. Based on this rename refactoring, the proposed approach suggested to rename its *set* method *setRedeliverDelay* (Line 6) in the same way, i.e., to replace 'redeliver' with 'redelivery'. The renaming recommendation was accepted because the method was renamed as expected in version 2.3.0 (Line 22). After that, the proposed approach suggested to rename the parameter *redeliverDelay* (Line 6) of this *set* method. Since this parameter is assigned directly to the field *redeliverDelay* and they have the exact same name in the original version (Line 7), they should be renamed consistently. However, the parameter has not been renamed accordingly (Line 22), and thus the renaming of the field and its *set* method is incomplete.

Another example is presented in Fig. 5. This example is similar to the one that is introduced in the preceding paragraph. The field *m_NumFolds* and its *set* method *setNumFolds* in version dev-3-1-9 have been renamed to *m_NumXValFolds* and *setNumXValFolds* in version dev-3-3 (Line 4 and Line 6), respectively. However, the parameter *newNumFolds* (Line 6) of the *set* method has not been renamed accordingly, which results in an incomplete renaming.

The third example is presented in Fig.6. The *get* method *getMaxProcessingTime* in vertion 1.3.0 was renamed to *getMaxProcessingTimeMillis* ('Millis' suggests that the processing time is counted in milliseconds) in version 1.4.0 (Line 7). However, the field *maxProcessingTime* (Line 5) that this method accesses was not renamed accordingly. The same is true for the field *minProcessingTime* (Line 4) and its *get* method (Line 10). These fields had better be renamed to make them consistent with their *get* methods.

4.6 Threats to Validity

A threat to external validity is that the evaluation was conducted on four applications only. Special characteristics of the applications may have biased the evaluation results. The reason why these applications were selected is discussed in Section 4.2. To make the conclusions more reliable, however, further evaluation on more applications should be conducted in the future.

A threat to construct validity is that the identification of rename refactorings might be inaccurate, and thus the measurements in Section 4.4 might be incorrectly calculated. The refactoring detectors used in the evaluation, i.e., *RefactoringCrawler* and *REF-FINDER*, might result in false positives and false negatives. To reduce the

```
/* version dev-3-3  */
public class ThresholdSelector extends DistributionClassifier implements OptionHandler {
        //...
  protected int m_NumXValFolds = 3;
        //...
  public void setNumXValFolds(int newNumFolds) {
    if (newNumFolds < 2) {
      throw new IllegalArgumentException("Number of folds must be greater than 1");
    }
    m_NumXValFolds = newNumFolds;
  }
       //...
}
```

Fig. 5. Incomplete Renaming in *Weka*

```
/* version 1.4.0  */
public class PerformanceCounter extends Counter {
        //...
    private double minProcessingTime = -1.0;
    private double maxProcessingTime;
        //...
    public synchronized double getMaxProcessingTimeMillis() throws Exception {
        return maxProcessingTime;
    }
     public synchronized double getMinProcessingTimeMillis() throws Exception {
        return minProcessingTime;
    }
        //...
}
```

Fig. 6. Another Incomplete Renaming in *Camel*

threat, participants manually checked files that *RefactoringCrawler* or *REF-FINDER* reported to contain potential rename refactorings. However, manual checking might miss refactorings as well. Files where *RefactoringCrawler* and *REF-FINDER* failed to identify any rename refactorings might contain real rename refactorings. These refactorings have been missed because participants had not manually checked such files (more than two hundred thousand files). Manually checking all such files is difficult, if not impossible. In the future, it would be interesting to evaluate the proposed approach on applications where renamings have been exactly recorded. On such applications, we do not have to identify rename refactorings anymore.

Another threat to construct validity is that the rename refactorings identified by *RefactoringCrawler* and *REF-FINDER* were rechecked by outsiders (three students) instead of the original developers of the subject applications. The check might be inaccurate because of lack of system knowledge and subjectiveness. To reduce the threat, three participants checked all of the potential renamings together. In case of diverging decisions, they discussed and voted if needed before the finial decision was made.

5 RELATED WORK

5.1 Identification of renaming opportunities

Abebe et al. [25] introduce the notion of *lexicon bad smells*. Lexicon bad smells indicate potential lexicon construction problems, and such lexicon problems often can be fixed by rename refactorings. Lexicon bad smells introduced by Abebe et al. [25] include *odd grammatical structure* (e.g., class identifiers without nouns), *term used to name both the whole and its parts*, *inconsistent identifier use*, *useless type indication*, and *identifer construction rules* (i.e., entity names that do not follow a standard naming convention). They also develop a tool, called *LBSDetectors* to identify such lexicon bad smells. Caprile and Tonella [26] propose an approach to standardize program identifier names. This approach involves two main steps. First, the lexicon (terms in identifier names) is standardized. For example, '*std*' should be standardized as 'standard'. In the second step, the arrangement of standard terms is standardized. For example, a method named '*PropagateDownHeap*' had better be renamed to '*HeapPropagateDown*' because the common syntactical pattern for method names is $<noun><verb><adverb>$ instead of $<verb><adverb><noun>$. Similar work to standardize identifier names has been conducted by Lawrie et al. [27], [28] and Butler et al. [29] as well.

Deissenboeck and Pizka [13] propose a model-based approach to identifying inconsistent naming. By building maps between concepts and names, the tool-supported *Identifier Dictionary* (IDD) developed by Deissenboeck and Pizka [13] can identify two categories of basic warnings. The first category of such warnings is given when two identifiers have identical name but different types. The second category of such warnings is given when an identifier is declared but never referenced. The first category of such warnings might suggest renaming opportunities. However, unique names of software entities might need renaming as well, and such renaming opportunities cannot be identified by *Identifier Dictionary*. Lawrie et al. [16] propose an approach to identify inconsistent naming as well. It differs from *Identifier Dictionary* [13] in that the latter requires an expert-constructed mapping from identifiers to concepts whereas the former does not. Thies and Roth [30] propose another way to identify inconsistent naming. They analyze variable assignments and identify variables that refer to the same object and are used in the same way. They suggest such variables to share the same name.

Høst and Østvold [31] propose an approach to identify naming bugs. They analyze the relationship between special terms, e.g., 'contain', in method names and special attributes of such methods, e.g. 'return boolean'. With such relationship, they infer some rules, e.g., *method like 'contain*' should return a boolean*. Methods that break these rules are reported as naming bugs.

De Lucia et. al. [32] suggest candidate identifier names by extracting terms from the text contained in requirements associated with the source code. The rationale is that terms in source code identifiers should be consistent with those in associated requirement documents.

As a conclusion, researchers have proposed a number of useful approaches to identifying badly named software entities, which suggests that people have recognized the necessity of facilitating identification of renaming opportunities. Our approach differs from existing approaches in that we identify renaming opportunities by expanding recently conducted rename refactoring whereas existing approaches identify badly named entities by analyzing lexicon or grammatical structure of their names [25], [26], [27], [28] or by checking inconsistency associated with identifiers [13], [16],[32].

Our approach is not meant to replace existing ones. Developers might get an initial set of renaming opportunities manually or semi-automatically by using one or more approaches introduced above. After that, she might find more renaming opportunities with our approach by expanding the initial ones.

5.2 Machine Learning in Refactoring Opportunity Identification

Maiga et al. [33] propose a support vector machine (SVM) based approach to identify anti-patterns that should be restructured. From given training data, their approach can learn how to identify similar anti-patterns. An advantage of their approach is that it does not depend on extensive knowledge of anti-patterns. They also use participants' feedback to improve accuracy by adopting an incremental SVM.

Khomh et al. [34] propose BDTEX (Bayesian Detection Expert) to detect anti-patterns. They build Bayesian Belief Networks based on *Goal Question Metric* [35], and calibrate the Bayesian Belief Networks with examples. Evaluation results on three anti-patterns, i.e., *Blob*, *Functional Decomposition* and *Spaghetti Code*, suggest that BDTEX is effective.

The approaches introduced in the preceding paragraph are useful in learning what kind of source code deserves refactorings. They learn from examples (e.g., conducted refactorings) and generate rules (or classifiers) for anti-pattern detection. As a result, they may work even if a formal definition of anti-patterns is not available. Similar to these approaches, our approach learns from conducted refactorings and works without a formal definition of bad names. Our approach differs from these approaches in that it learns from a single conducted refactoring, and applies what it has learned from this example to a small number of entities that are closely related to the refactoring. Our approach does not result in generic rules because it assumes that new refactorings should share the same context with the conducted refactoring. In contrast, the approaches introduced in this section require a large amount of related examples to learn universal rules or classifiers that can be used in different applications or different parts of the same application.

5.3 Detection of Rename Refactorings

Xing and Stroulia [19] propose UMLDiff to detect refactoring (including rename refactoring) history by comparing class diagrams of two successive versions of the same application. Dig et al. [20] propose RefactoringCrawler to detect refactoring history by comparing Java source code of two successive versions of the same application. Weissgerber and Diehl [21] detect refactorings by comparing signatures of software entities, and thus they can identify *rename method* and *rename class* refactorings. Prete et al. [23] propose a template-based approach, called REF-FINDER, to detect refactorings. Kawrykow et al. [36] propose DIFFCAT to identify non-essential changes (including rename refactorings) in version histories. These approaches can be used to identify rename refactorings although they are designed to detect refactorings in general.

Approaches specially designed to detect rename refactorings are also available. Malpohl et al. [18] propose *renaming detector* to detect rename refactorings. Arnaoudova et al. [11] propose an approach, called REnaming Program ENTitles (REPENT), to identify rename refactorings across different versions of a program and to automatically classify such refactorings. They also implement the approach as a series of scripts.

These algorithms and tools differ from our approach in that they identify conducted rename refactorings whereas we identify software entities that should be renamed. IDEs might use these algorithms and tools to discover recently conducted rename refactorings (especially manually conducted refactorings), and then use our approach to identify more renaming opportunities by expanding the discovered rename refactorings.

6 Conclusions and Future Work

In this paper we propose an approach to identify renaming opportunities by expanding conducted rename refactorings. It avoids difficult natural language understanding and difficult semantic analysis. It also avoids subjective factors, e.g., preference of software engineers. To the best of our knowledge, it is the first one that identifies new refactoring opportunities by expanding conducted ones. The proposed approach has been evaluated on four open-source applications. Our evaluation results show that it is accurate in recommending renaming opportunities and in recommending new names for badly named entities. Our evaluation results also show that a substantial percentage of rename refactorings can be expanded by the proposed approach.

Basically, this work applies the principle: if a software entity is renamed, other entities that look similar should be consistently renamed. However, there might be many exceptions to this principle. Suppose two methods in different classes were given the same method name, but after that they evolved in completely different ways. In this case, if one of them is renamed, it is likely that the other should not be renamed in the same way. Consequently, it should be interesting to investigate in the future to what extent the evolution histories of different entities can help to avoid such kind of false positives. It should be interesting as well to investigate how to compute the strength of recommendation. For example, if the past history highlights that the developer has done several of such renaming refactorings and many more could be done, we can reward the renaming recommendation by assigning a greater strength indication to it.

In Section 4 the proposed approach has been evaluated on the history of open-source applications, and the objective and quantitative evaluation results (metrics) are promising. However, there is no evaluation of 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). To facilitate subjective evaluation by other developers or researchers, we make the prototype implementation (an Eclipse plug-in called *Rename Expander*) of the approach publicly available[10] (including the source code).

In the future, it would be interesting to apply the key ideal of this paper, i.e., identifying new refactoring opportunities by expanding conducted ones, to other refactorings, e.g., *extract method* and *move method*. According to our software development experience, it is likely that more than one method should be moved when system functionality is redistributed (restructuring). By investigating the relationship among such methods that are moved together, we might design approaches to identify which methods should be moved and where they should be moved whenever a *move method* refactoring is conducted. These approaches might help to avoid incomplete restructuring.

10. http://www.sei.pku.edu.cn/~liuhui04/tools/rename/

Acknowledgments

The authors would like to say thanks to the associate editor and the anonymous reviewers for their valuable suggestions.

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] 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.

[4] 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.

[5] 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.

[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] 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

[8] E. Mealy and P. Strooper, "Evaluating software refactoring tool support," in *Australian Software Engineering Conference (ASWEC'06)*, April 2006.

[9] G. Bavota, S. Panichella, N. Tsantalis, M. D. Penta, R. Oliveto, and G. Canfora, "Recommending refactorings based on team comaintenance patterns," in *the 29th IEEE/ACM International Conference on Automated Software Engineering (ASE 2014)*, September 15-19 2014. [Online]. Available: http://users.encs.concordia.ca/~nikolaos/publications/ASE_2014.pdf

[10] 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.

[11] V. Arnaoudova, L. Eshkevari, M. Penta, R. Oliveto, G. Antoniol, and Y.-G. Gueheneuc, "Repent: Analyzing the nature of identifier renamings," *Software Engineering, IEEE Transactions on*, vol. 40, no. 5, pp. 502–532, May 2014.

[12] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "What's in a name? a study of identifiers," in *14th IEEE International Conference on Program Comprehension (ICPC 2006)*, June 2006, pp. 3–12.

[13] F. Deissenboeck and M. Pizka, "Concise and consistent naming," *Software Quality Journal*, vol. 14, no. 3, pp. 261–282, 2006. [Online]. Available: http://dx.doi.org/10.1007/s11219-006-9219-1

[14] A. Takang, P. A. Grubb, and R. D. Macredie, "The effects of comments and identifier names on program comprehensibility: an experiential study," *Journal of Program Languages*, vol. 4, no. 3, pp. 143–167, 1996.

[15] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "Effective identifier names for comprehension and memory," *Innovations in Systems and Software Engineering*, vol. 3, no. 4, pp. 303–318, 2007. [Online]. Available: http://dx.doi.org/10.1007/s11334-007-0031-2

[16] D. Lawrie, H. Feild, and D. Binkley, "Syntactic identifier conciseness and consistency," in *Source Code Analysis and Manipulation, Sixth IEEE International Workshop on*, Sept 2006, pp. 139–148.

[17] X. Ge, Q. DuBose, and E. Murphy-Hill, "Reconciling manual and automatic refactoring," in *Software Engineering (ICSE), 34th International Conference on*, June 2012, pp. 211–221.

[18] G. Malpohl, J. Hunt, and W. Tichy, "Renaming detection," in *Automated Software Engineering, The Fifteenth IEEE International Conference on*, 2000, pp. 73–80.

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

[20] 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

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

[22] 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

[23] 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.

[24] 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.

[25] S. Abebe, S. Haiduc, P. Tonella, and A. Marcus, "Lexicon bad smells in software," in *16th Working Conference on Reverse Engineering (WCRE '09)*, Oct 2009, pp. 95–99.

[26] B. Caprile and P. Tonella, "Restructuring program identifier names," in *Software Maintenance, Proceedings. International Conference on*, 2000, pp. 97–107.

[27] D. Lawrie and D. Binkley, "Expanding identifiers to normalize source code vocabulary," in *Software Maintenance (ICSM), 27th IEEE International Conference on*, Sept 2011, pp. 113–122.

[28] D. Lawrie, D. Binkley, and C. Morrell, "Normalizing source code vocabulary," in *Reverse Engineering (WCRE), 17th Working Conference on*, Oct 2010, pp. 3–12.

[29] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Mining java class naming conventions," in *Software Maintenance (ICSM), 27th IEEE International Conference on*, Sept 2011, pp. 93–102.

[30] A. Thies and C. Roth, "Recommending rename refactorings," in *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, ser. RSSE '10. New York, NY, USA: ACM, 2010, pp. 1–5. [Online]. Available: http://doi.acm.org/10.1145/1808920.1808921

[31] E. W.Høst. and B. M.Østvold, "Debugging method names," in *23rd European Conference on Object-Oriented Programming (ECOOP 2009)*, ser. Lecture Notes in Computer Science, S. Drossopoulou, Ed. Springer Berlin Heidelberg, 2009, vol. 5653, pp. 294–317. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03013-0_14

[32] A. De Lucia, M. Di Penta, and R. Oliveto, "Improving source code lexicon via traceability and information retrieval," *Software Engineering, IEEE Transactions on*, vol. 37, no. 2, pp. 205–227, March 2011.

[33] A. Maiga, N. Ali, N. Bhattacharya, A. Sabane, Y. Gueheneuc, and E. Aimeur, "Smurf: A svm-based incremental anti-pattern detection approach," in *Reverse Engineering (WCRE), 19th Working Conference on*, Oct 2012, pp. 466–475.

[34] F. Khomh, S. Vaucher, Y.-G. Guhneuc, and H. Sahraoui, "Bdtex: A gqm-based bayesian approach for the detection of antipatterns," *Journal of Systems and Software*, vol. 84, no. 4, pp. 559 – 572, 2011, the Ninth International Conference on Quality Software. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121210003225

[35] V. Basili and D. Weiss, "A methodology for collecting valid software engineering data," *Software Engineering, IEEE Transactions on*, vol. SE-10, no. 6, pp. 728–738, Nov 1984.

[36] D. Kawrykow and M. P. Robillard, "Non-essential changes in version histories," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 351–360. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985842