A Survey on Renamings of Software Entities

GUANGJIE LI, HUI LIU, and ALLY S. NYAMAWE, School of Computer Science and Technology, Beijing Institute of Technology, China

More than 70% of characters in the source code are used to label identifiers. Consequently, identifiers are one of the most important source for program comprehension. Meaningful identifiers are crucial to understand and maintain programs. However, for reasons like constrained schedule, inexperience, and unplanned evolution, identifiers may fail to convey the semantics of the entities associated with them. As a result, such entities should be renamed to improve software quality. However, manual renaming and recommendation are fastidious, time consuming, and error prone, whereas automating the process of renamings is challenging: (1) It involves complex natural language processing to understand the meaning of identifers; (2) It also involves difficult semantic analysis to determine the role of software entities. Researchers proposed a number of approaches and tools to facilitate renamings. We present a survey on existing approaches and classify them into identification of renaming opportunities, execution of renamings, and detection of renamings. We find that there is an imbalance between the three type of approaches, and most of implementation of approaches and evaluation dataset are not publicly available. We also discuss the challenges and present potential research directions. To the best of our knowledge, this survey is the first comprehensive study on renamings of software entities.

CCS Concepts: • **Software and its engineering** → **Software creation and management**; **Software post-development issues**; **Software reverse engineering**;

Additional Key Words and Phrases: Rename refactoring, software quality, identifier

ACM Reference format:
Guangjie Li, Hui Liu, and Ally S. Nyamawe. 2020. A Survey on Renamings of Software Entities. *ACM Comput. Surv.* 53, 2, Article 41 (April 2020), 38 pages.
https://doi.org/10.1145/3379443

1 INTRODUCTION

1.1 Renamings of Software Entities

Identifiers are the names of software entities in source code, e.g., variables, attributes, methods, classes, parameters, and so on [57]. They account for about 70 percent of source code characters [49]. Meaningful identifiers can reveal the role of the related entities and form a concise description of the program behavior [20, 153]. Such identifiers should capture the domain-specific knowledge that programmers want to express when writing code [17]. Many studies demonstrated that natural language information embedded in identifiers are important source for program

The work is partially supported by the National Natural Science Foundation of China (61772071, 61690205).
Authors' address: G. Li, H. Liu (corresponding author), and A. S. Nyamawe, School of Computer Science and Technology, Beijing Institute of Technology, Beijing, China, 100081; emails: {3120150428, liuhui08}@bit.edu.cn, nyamawe@udom.ac.tz.

0360-0300/2020/04-ART41 $15.00
https://doi.org/10.1145/3379443

comprehension and can be used to decrease the expenses of software maintenance [57, 158]. Consequently, identifiers are crucial to understand and maintain the source code [10].

Proper identifiers should be readable for developers [110, 167], match the role of their corresponding entities [125], be consistent to the program context [49], and comply with particular naming schemes [15, 35, 38, 69, 88]. The interpretation of identifiers is often context dependent. For example, identifier "DiZhi" is non-sensical in most context, but in the context of Chinese developers, it may make perfect sense and not require renaming.

However, in many cases, it is challenging to name software entities properly, and as a result, *improper identifiers* are widespread [16, 161]. The reasons of improper identifiers are presented as follows:

- First, inexperienced developers may fail to find proper names. On the one hand, the role of the software entity is complex, which makes it challenging to describe with a few words. On the other hand, long names may reduce the readability of source code [106]. Consequently, developers must select a few dictionary words, abbreviated non-dictionary words, or domain-specific words to compose a short identifier that comply with the contextual naming convention and reveal the key role of the named entity.
- Second, developers may compose identifiers casually because of constrained schedule. Such casual identifers are often not representative of the role and not well thought-out.
- Third, different developers may pick up the same name for different entities, or different names for the same entity.
- Finally, software evolution may make meaningful identifiers obsolete. During evolution, software entities may be merged or split. As a result, the original names of the entities may not match the role of the new evolved entities any more.

Improper identifiers decrease the readability and maintainability of source code and also lead to software defects [11, 28, 32]. Consequently, *rename refactoring* is commonly applied to rename improperly named software entities with proper names. The rename refactoring allows the concrete execution of renamings. In general, we refer to all related steps involved in the process of rename refactoring (i.e., deciding where and how to name a software entity and actually changing related identifiers) as *renamings*. We refer to the actual renaming of entities (i.e., propagating the renaming in such a way that the program still compiles and runs after the renaming) as *execution of renamings*.

Renamings are widely used by developers. A survey performed by Arnaoudova et al. [20] indicated that 39% of 71 developers perform renamings almost every day (at least a few times per week), 46% of developers perform renamings several times a month, 14% of developers perform renamings monthly, and only 1% of developers never perform renamings. Another survey conducted by Murphy et al. [128] confirmed the popularity of renamings. All of the 99 participants in the survey (including application developers, academic researchers, system architectures, and chief information officers) performed renamings. The popularity of renamings suggests that software entities are often improperly named and must be renamed.

1.2 Typical Process of Renamings

The typical process of renaming is presented in Figure 1. The first step of renaming is identification of renaming opportunities, which aims to identify improper names from source code. The second step of renaming is recommendation of new names for improper named entities. In practice, the identification of renaming opportunities is combined with recommendation of proper names. Consequently, in the following sections, we will describe these two aspects together. The third step of renaming is execution of renaming, which actually replaces improper named entities

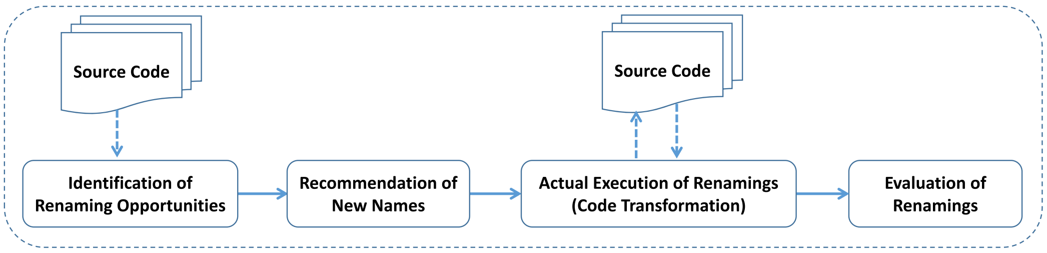

Source Code
Source Code
Identification of Renaming Opportunities
Recommendation of New Names
Actual Execution of Renamings (Code Transformation)
Evaluation of Renamings

Fig. 1. Typical process of renaming.

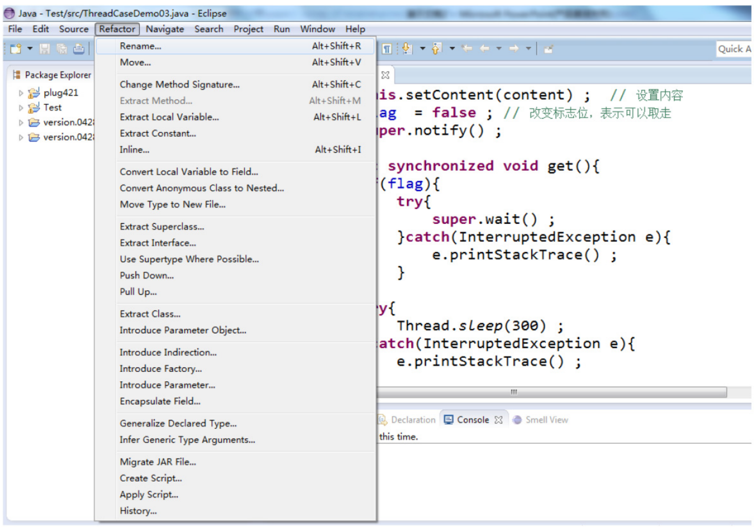
with recommended new names and propagates the renaming in such a way that the program still compiles and runs after the renaming. Corner cases invisible to the renaming approach may result in compilation or runtime errors. Generally, the objective of this step is to preserve name bindings, although some code (e.g., in *Unison* [2]) may be identified by its hash. Finally, the last step of renaming is evaluation of renaming, which checks/validates the correctness and performance of conducted renaming. Although, in this article, we surveyed approaches to detecting renamings, such approaches are not directly related to the typical process of renamings.

1.3 Automation of Renamings

It is time consuming, fastidious, and error prone to manually conduct renamings. First, when developers decide to rename a software entity, they would need to rename all of its references consistently. It is fastidious to change all occurrences one by one. Second, the references to a certain name may appear in different places or different files. Forgetting to rename any occurrence may lead to compilation and runtime errors. Finally, it may take time to identify all improper names manually, especially when the program is large. Identification of improper names could be simple and straightforward if renamings are conducted as developers program. In such cases, developers know exactly where and how to rename entities. However, sometimes developers do dedicate time to software refactoring (e.g., renaming). According to Soares et al. [162] and Murphy et al. [130], developers sometimes allocate a duration for refactorings only, and around 17% of the developers perform renaming as a stand-alone activity [20]. In such cases, automated identification of improper identifiers could be helpful.

Automation of renamings is beneficial because refactoring tools are faster and less error prone [130]. To facilitate the process of renamings, mainstream IDEs (e.g., Eclipse, Visual Studio, Intellij IDEA, Netbean, and Borland JBuilder) provide automated tool support for renamings. Murphy et al. [128] reported that renaming is one of the most widely used functionalities in current IDEs. Burgula and Reddy [27] found that renaming is one of the most popular refactorings that are supported by IDEs. A survey conducted by Eshkevari [56] suggests that 72% of developers employ automatic tool support to perform renamings. Another survey [122] on 139 professional developers suggests that renaming is the most widely provided functionality by development tools, besides identifiers highlighting. IDEs even put the *rename* menu item at the top of the *refactoring* menu because renaming is more popular than other refactorings. Figure 2 shows the *refactoring* menu of Eclipse, where the *Rename...* menu is set at the top.

Although tools can improve the speed and accuracy of software maintenance, refactoring tools are not used as much as they could be [130]. Refactoring tools integrated in mainstream IDEs (e.g., Eclipse) can rename the selected entity and propagate changes to other files in the current project. They highlight and update all references to the selected entity in real time when developers type in new names. Such refactoring tools significantly advanced the state of the practice in renamings.



```
is.setContent(content) ;  // 设置内容
ag  = false ; // 改变标志位，表示可以取走
per.notify() ;

 synchronized void get(){
(flag){
  try{
      super.wait() ;
  }catch(InterruptedException e){
      e.printStackTrace() ;
  }

y{
  Thread.sleep(300) ;
atch(InterruptedException e){
  e.printStackTrace() ;
```

```
this time.
```

Fig. 2. Refactor menu in Eclipse.

However, even the most widely used refactoring frameworks in current IDEs cannot automate the whole process of renamings completely and correctly [152]. Developers often manually identify entities that should be renamed and manually find new names for the involved entities. A questionnaire suggested that only 2 out of 16 students used refactoring tools in an object-oriented programming class [129].

The main challenges in automating renamings are explained as follows:

- It is impossible to update all references to the renamed entity, especially for language that supports *reflection*, e.g., Java. Failing to update any of them may result in compilation and runtime errors.
- It is challenging to recommend proper new names. Proper names should reveal the role of software entities concisely, be consistent to the context, and comply with the naming conventions of the software organization and programming language.
- It is challenging to recognize improper names in a given software. On the one hand, it is challenging to mine the information embedded in an identifiers because it may involve complex natural language processing (NLP). On the other hand, it is challenging to understand automatically the meaning of software entities because it would require complex semantic analysis of source code.

Researchers proposed different approaches and tools to facilitate renamings. Automatic tools may fail to uncover all improper identifiers, but they can successfully and accurately identify some specific one. Deissenboeck and Pizka [49] identify inconsistent identifiers based on formal analysis of identifiers properties. Lawrie et al. [104] propose another approach to identifying inconsistent identifiers based on syntactic containment rules. Høst and Østvold [79] proposed an automatic approach to identifying inconsistencies between method names and method implementations based on predefined implementation rules. In this article, we take a survey to find out the full picture of papers concerning renaming identification, execution, and detection. We also present comprehensive analysis on different approaches.

1.4 Methodology

To investigate the state-of-the-art researches on renamings, we conduct the search and selection according to the following process. First, we conduct an initial pilot search for the relevant papers from the well-known digital libraries including Google Scholar, ACM Digital Library, and IEEE Xplore Digital Library. We query the aforementioned digital libraries using the following keywords individually: "rename refactoring", "identifier renaming", "naming bug", "restructure identifier", "rename detection", "identifier splitting", and "abbreviation expansion". For example, we formulate the following query string to search ACM Digital Library:

```
"query": {(rename refactoring)}
"filter": {"publicationYear": {"gte": 2000}},
{owners.owner=HOSTED}
```

We perform the search process in each library on each keyword aforementioned and take the top 100 papers sorted by relevance for each query. Since some queries return less than 100 relevant papers, we get 2,051 candidate papers by the initial pilot search.

Second, for each candidate paper, we perform manual search to classify each paper as either included or excluded by checking abstract and title. We define the following inclusion criteria which constrain the scope of this survey:

- The main objective of this article is to solve one of the problem in the process of renamings, e.g., to identify improper names, to recommend proper names, or to execute renamings.
- Research papers that focus on facilitating renamings, e.g., preprocessing of identifiers, detection or analysis of conducted renamings.

After eliminating duplicate papers, 49 papers are included by criterion one, and 40 papers are included by criterion two. Therefore, after applying the stated inclusion criteria, we get 89 candidate papers.

Finally, we read through the full papers and expand manual search by forward and backward snowballing. Such process involves searching each library aforementioned on the title of each paper iteratively against the same inclusion criteria until no more relevant works are found. One of the expansion techniques is to go through the reference section of the initial set of papers to explore the relevant cited papers and further subject them for review. At the end of this stage, we find 20 relevant papers. As a result, we get 109 relevant papers to formulate the dataset of this survey. We further analyze the distribution of the selected papers according to the publication venues. Figure 3 shows venues where at least two selected papers have been published. We find that the *ASE* conference published the highest number of renaming papers.

We further classify the selected papers into different taxonomy according to the problems that they are addressing, i.e., the different steps (tasks) they support in the process of renamings. This taxonomy formulates the basic structure of this survey. If a given paper contributes on more than one tasks of renamings, we discuss it in different sections (each section is allocated for a specific renaming task) regarding its contributions to different tasks. An overview of the taxonomy is presented in Figure 4. The taxonomy also leverages additional dimensions besides the functionality of the related work. Such additional dimensions include the techniques used by the approach, the information on which it builds, its advantage/disadvange, the dataset on which it was evaluated, evaluation method, the performance it achieves, and its availability. Figure 4 also shows an imbalance between the three main types of approaches, i.e., 65% (=71/109) of surveyed papers focus on identification of renaming opportunities, 18% (=20/109) focus on execution of renamings, and 17% (=18/109) focus one detection of renamings. A complete classification of the papers according to the taxonomy is presented as tables in the online appendix [4].

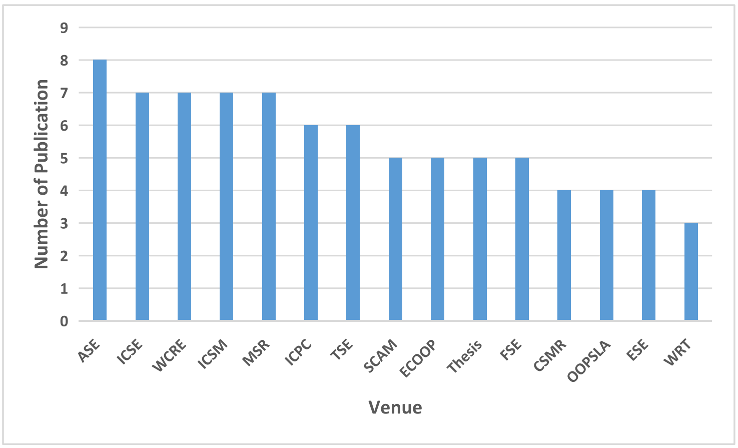


Fig. 3. Number of publications per venue.

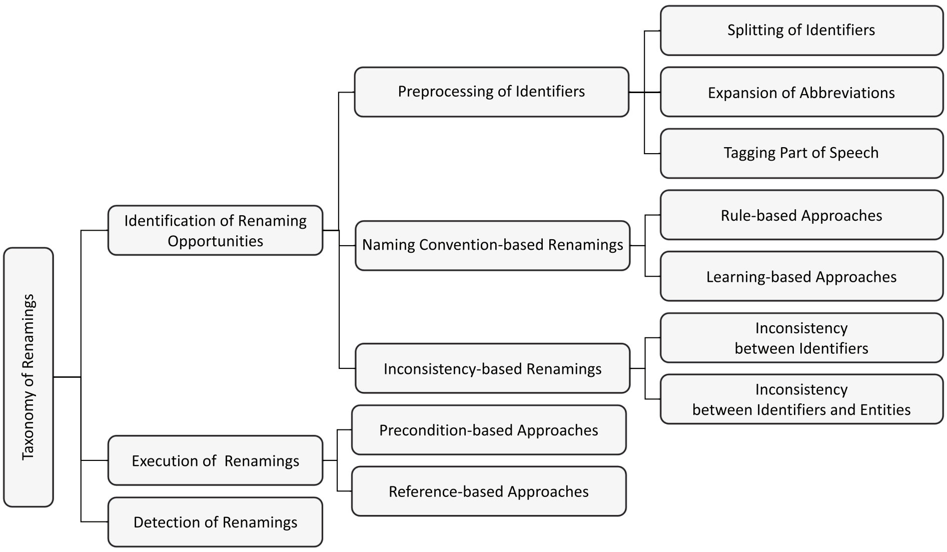

Taxonomy of Renamings
Identification of Renaming Opportunities
Execution of Renamings
Detection of Renamings
Preprocessing of Identifiers
Naming Convention-based Renamings
Inconsistency-based Renamings
Precondition-based Approaches
Reference-based Approaches
Splitting of Identifiers
Expansion of Abbreviations
Tagging Part of Speech
Rule-based Approaches
Learning-based Approaches
Inconsistency between Identifiers
Inconsistency between Identifiers and Entities

Fig. 4. Overview of researches on renamings.

The rest of the article is structured as follows: Section 2 introduces the preprocessing of identifiers. Section 3 presents researches on identification of renaming opportunities based on naming conventions. Section 4 illustrates researches on identification of renaming opportunities based on inconsistencies. Section 5 presents researches on the execution of renamings. Section 6 presents researches on the detection of renamings. Section 7 concludes the survey and introduces future work.

2 PREPROCESSING OF IDENTIFIERS

As mentioned earlier, identifiers preprocessing is one of the key steps in identification of renaming opportunities. Often, developers construct identifiers with multiple words, abbreviations, and acronyms [33, 55]. Consequently, the first step of identifier preprocessing is to split identifiers into constituent words [55]. After that, the second step of identifier preprocessing is to expand component abbreviations and acronyms into full words or phrases they actually represent to help program comprehension. Finally, the last step of identifier preprocessing is to tag the part of speech (POS) of component words to infer linguistic relationships and syntactic information embedded in the word sequences.

However, it is challenging to preprocess identifiers for the following reasons: (1) it is difficult to split identifiers into component words because often no explicit words division markers are used to separate component words; (2) it is difficult to expand abbreviations and acronyms because a given abbreviation or acronym often has multiple expansions from which to choose; and (3) it is difficult to tag the part of speech of component words because the word sequences extracted from identifiers rarely construct a grammatically correct sentence and natural language POS taggers cannot be used to tag them.

Consequently, different approaches have been proposed to preprocess identifiers, i.e., splitting of identifiers, expansion of abbreviations, and part of speech tagging. In this section, we first introduce some terminologies relevant to identifiers preprocessing and provide detailed information of the existing approaches.

2.1 Terminologies

An identifier is composed of a single or multiple dictionary words, abbreviations, and acronyms. A *dictionary word* refers to a natural language word listed in a publicly available English dictionary or software dictionary predefined by organizations/developers. Whereas, *non-dictionary words* are words not in any dictionaries. An *abbreviation* is the shortened form of a word (e.g., “str” stands for “string”). An *acronym* is a word defined by the author/developer that is formed from the initial letters of a nominal group (e.g., “RAM” stands for “Random Access Memory”). Dictionary words, known abbreviations, and known acronyms composing identifiers are called *soft words. Word separators* (e.g., underscore), also known as *delimiters*, are called *division markers.* However, since developers do not always separate all component words with explicit *division markers* conventionally, strings delimited by division markers, called *hard words* [59], may contain a single or multiple soft words. For example, the identifier *initialize_linkstack* contains a division marker *underscore* and two hard words *initialize* and *linkstack.* The hard word *initialize* is composed of a single soft word (i.e., *initialize*), whereas the hard word *linkstack* is composed of two soft words (i.e., *link* and *stack). Tokens* are sequences of elements separated by division markers forming up identifiers. *Lexical analysis of identifiers* is the process of identifying the words composing identifiers, *syntax analysis of identifiers* is the process of analyzing the syntactical structure of component words in constructing identifiers, and *semantic analysis of identifiers* is the process of analyzing the associated meaning with identifiers [37].

2.2 Splitting of Identifiers

As mentioned earlier, splitting of identifiers, also known as *lexical analysis*, *identifier tokenization*, or *identifier segmentation*, aims to partition program identifiers into component words [81]. Unlike traditional natural language text, words in identifiers are not delimited by usual delimiters like space. Java developers often create identifiers according to camel case convention [71], whereas C programmers often concatenate words using an underscore as separator. It is simple and straightforward to split identifiers where explicit division markers (e.g., capital letter or special separator) are used to separate component words. For example, it is trivial to split identifier “new_entry” into the hard words “new” and “entry” according to the division marker underscore. However, explicit division markers are not followed by developers universally. Enslen et al. [55] found that 6% of multi-word identifiers in Java cannot be split according to camel case convention, i.e., 6% of hard words are made up of multiple soft words. Binkley and Lawrie [23] found that 7.5% of Java hard words are composed of multiple soft words, whereas 16.5% of C hard words are composed of multiple soft words. Butler et al. [33] found that 9% of identifiers either concatenate multi-words in same-case or contain digits. Usually, it is challenging to split identifiers that do not exploit explicit division markers or contain digits because of the following reasons:

- First, ambiguity arises when identifiers consist of same-case words. For example, it is challenging to split identifier "newstop" because there is no division marker between the component soft words, "news top" and "new stop" are both potential correct splitting.
- Second, there is no simple rule to split identifiers that contain digits. The digits may be part of an acronym (e.g., "POP3Protocol" stands for "POP3 Protocol"), or it may be a discrete part of the identifier (e.g., "Word2Vector" stands for "Word to Vector").
- Third, it is difficult to split mix-case identifiers which contain a sequence of upper case letters (e.g., *SQLLite*).

To accurately split identifiers, researchers have proposed different approaches to resolve ambiguity. According to the techniques exploited, we classify existing identifier splitting approaches into three categories, i.e., heuristic-based approaches, learning-based approaches, and graph-based approaches. A summarization of such splitting approaches is presented in Table A.1 in the appendix.

2.2.1 Heuristic-Based Approaches. Heuristic-based approaches recursively look for dictionary words or abbreviations according to predefined heuristic rules until a predefined condition is satisfied. *Greedy* algorithm [59] first splits identifiers into a sequence of hard words based on predefined division markers (e.g., underscore). Second, it looks up hard words in three dictionary lists, i.e., *ispell* dictionary [149], known abbreviations list, and stop words list. The stop words list includes keywords (e.g., "for"), predefined variables (e.g., "NULL"), and library functions (e.g., "strcpy"). If the hard word appears in one of the three lists, *Greedy* takes it as a soft word and stops splitting on it further. If it does not appear in any lists, *Greedy* will search the dictionaries for the longest *prefix* and *suffix* of the hard word as a soft word and remove it from the hard word. If no soft word is found by the *prefix* and *suffix* search, the *Greedy* algorithm replaces the first character from the identifier repeatedly with the division marker. It repeats the process iteratively on the remaining string until the remaining string is empty or no dictionary words are included in the remaining string. If the *suffix* and *prefix* searches return different sequences of soft words, the one with higher ratio of soft words found in the lists to the total soft words found is selected as the correct split. The evaluation of the approach involves 4,000 identifiers randomly selected from 186 programs (C, C++, Java, and Fortran) and manually labeled by four programmers. The evaluation result suggests that the accuracy of *Greedy* algorithm ranges from 75% to 81%.

Instead of requiring a predefined dictionary, Enslen et al. [55] propose a frequency-based approach, *Samurai*,[1] to incorporate evolving terminologies automatically. *Samurai* hypothesizes that soft words used in one identifier are more likely to be used in other identifiers in the same program or other programs. Consequently, it splits identifiers by mining word frequencies in source code. It first builds two tables to store the frequencies of words in the current program and the frequencies of words in a global-source code corpus by exploiting a conservative camel case splitting algorithm. Second, the approach splits identifiers into a sequence of digits, all same-case substrings, and mix-case strings by inserting division markers around the digit sequence and alternating cases. Finally,

[1]http://tools.soccerlab.polymtl.ca/samurai/index.php.

it determines whether to split each same-case and mix-case string into two substrings recursively, based on the following function and selects the sequence of words with the highest score as the correct split:

$$score(w, p) = pFr(w, p) + gFr(w)/log_{10}(allFr(p)), \tag{1}$$

where *pFr*(*w,p*) refers to the frequency of word *w* in the program *p*, *gFr*(*w*) refers to the frequency of *w* in the global source code corpus, *allFr*(*p*) refers to the frequency of all words in the program *p*. Compared to *Greedy* algorithm, *Samurai* makes fewer over-split and is specialized in mix-case splitting problem. It takes digits into consideration whereas isolating them at early stage. The evaluation of the approach builds the program-specific and global frequency tables using 9,000 open-source Java programs downloaded from SourceForge[2]). Evaluation on 8,466 identifiers randomly selected and manually inspected by two annotators suggests that the accuracies of *Samurai* and *Greedy* are 97% and 95%, respectively.

Inspired by *Samurai*, Hucka [81] also uses word frequencies to determine splitting policy. However, instead of mining word frequencies from local program, the author proposes a Python toolkit *Spiral*[3] to split identifiers by exploiting only a global table of word frequencies obtained by mining 46,000 Python source code repositories. *Spiral* is implemented based on Python Natural Language Toolkit (*NLTK*)[4] [116]. Similarly, Sureka [165] exploits word frequency (popularity) computed by Yahoo Web search engine to determine the split position. With no need of predefined dictionaries, the approach takes the number of hits clicked on words or the results of words searching returned by Yahoo Web search engine as the proxy of word popularity, conceptual correlations, and semantic relatedness. Evaluation on 500 identifiers from the publicly available dataset *INVocD*[5] [34] suggests that the accuracy of this approach is around 30%.

TIDIER [71, 117, 118] overcomes the limitation of existing approaches in splitting identifiers containing abbreviations. It specializes in recognizing transformed words from identifiers by computing the minimal string-edit distance between transformed words in the identifiers and words in predefined thesaurus (including generic English dictionary, domain-specific program dictionary, and high-level documentation contextual dictionary) using Speech Recognition techniques. *TIDIER* first uses word transformation rules and a hill-climbing algorithm to split identifiers into a number of words. Second, it computes the Levenshtein distance [108] between words in identifiers and words in the dictionary using DTW (Dynamic Time Warping) [134]. Finally, it selects the split having minimum Levenshtein distance as the correct split. *TIDIER* algorithm is dictionary dependent and tends to be slow when the dictionary size increases because DTW algorithm computes edit distance between words in cubic time. The evaluation of the approach involves 1,026 identifiers randomly extracted from 340 C programs and manually inspected by two of the authors. Evaluation result suggests that the accuracy of *TIDIER* is 54%, compared to 31% of *Samurai*.

Instead of applying transformation rules, *GenTest* algorithm [103] generates all possible splits and selects an optimal split based on a set of metrics (e.g., soft word characteristics, external information, and internal information). The soft-word metrics prefer the split with fewer soft words, longer soft words, fewer vowel-free soft words, fewer single-letter soft words. The external metrics prefer the split with larger number of soft words found in the dictionary, more soft words found in program specific dictionary, and more co-occurrence information found in Google dataset. The internal metrics prefer soft words more frequently used in the source code. *GenTest* exploits a logic regression model to capture the best combination of metrics. Because it generates all possible splits for each identifier, it is time-consuming (exponential number) for *GenTest* to split longer

[2]https://sourceforge.net/.
[3]https://github.com/casics/spiral.
[4]https://www.nltk.org/.
[5]http://oro.open.ac.uk/36992.

identifiers. The evaluation of *GenTest* involves 4,000 identifiers randomly selected from 186 programs (including C, C++, Java, and Fortran). Half of the dataset are used for model construction, half for model validation. The identifiers are manually split by four programmers. Result suggests that the accuracy of *GenTest* is 82%, compared to 64% and 70% of *Greedy* and *Samurai*, respectively.

INTT[6] [29, 31, 33] improves the accuracy of splitting identifiers containing same-case words and digits. It first splits mix-case identifiers or those containing digits using heuristic rules based on *SCOWL* [157] dictionary, abbreviations dictionary, and acronyms dictionary. Second, for each same-case substring, it applies a greedier algorithm to iterate over the substring forward and backward. Compared to the *Greedy* algorithm [59], the greedier algorithm does not require that the input substring to begin or end with known words. The evaluation of *INTT* involves 4,000 identifiers randomly selected from 60 Java projects. The identifiers are manually split by the first author. Result suggests that the accuracy of *INTT* is 96%–97%.

Hill et al. [76] build up a publicly available dataset and compare the *Greedy*, *Samurai*, *GenTest*, *INTT*, and *TIDIER* algorithms. To build up the dataset, they first extract identifiers from 2,117 open-source programs (C, C++, Java, and Fortran) and randomly select 4,000 identifiers from each language. Second, after removing duplicates across languages and getting consensus from around 100 annotators, the final dataset contains 2,663 identifiers. Evaluation on the dataset suggest that: (1) *GenTest* and *Samurai* perform the best over all identifiers; (2) *TIDIER* performs better if identifiers are pre-split into hard words according to camel case convention; (3) for identifiers with no division markers, *Greedy*, *GenTest*, and *TIDIER* algorithms with small dictionary perform the best; and, (4) for same-case splits, *Samurai*, *GenTest*, and *TIDIER* perform the best.

2.2.2 Learning-Based Approaches. Existing two learning-based approaches learn the probabilities of inserting division markers in different positions of identifiers by training neural network model and *n-gram* language model, respectively.

Feild et al. [59] split hard words into soft words using Fast Artificial Neural Network (FANN) [137]. The approach first splits each identifier into hard words using division markers. Second, it encodes each character of hard word into a numeric value, feeds the numeric values into the neural network, trains the weight value of each *FANN* node using improved resilient back propagation algorithm, and represents the probabilities of inserting word separators at different positions of a hard word with the output value of each position. To evaluate the approach, the authors train and test five separate neural networks with 4,000 identifiers randomly selected from 186 programs (C, C++, Java, and Fortran) using the standard 9 to 1 ratio. The identifiers are manually inspected by four programmers. The results suggest that the accuracy of the approach ranges from 71% to 95%. The approach outperforms the *Greedy* algorithm [59] only when the training and testing data are labeled by the same programmer.

Pirapuraj and Perera [144] split abbreviated and concatenated words in identifiers based on *n-gram* language model and WordNet [62]. They first use *2-gram* to split identifiers and check words with Stanford SpellChecker and WordNet. If SpellChecker (or WordNet) recognizes the word, then the algorithm removes it from the identifier and repeats the process on the remaining string until the remaining string is empty. If SpellChecker can not recognize the word, the proposed approach increases the *n* value in the *n*-gram model and repeats the process until *n* reaches the length of the remaining string. The evaluation of the approach involves 472 conventional identifiers (having explicit division markers) and six unconventional identifiers from five Java projects. The evaluation result suggests that the accuracy of this approach in splitting conventional identifiers and unconventional identifiers are 94% and 87%, respectively.

[6] http://oro.open.ac.uk/28352/.

2.2.3 *Graph-Based Approaches.*

Graph-based approaches treat identifier splitting as a path optimization problem. They first transform programs into graph representations, then map the problem of identifier splitting into the problem of searching shortest path in weighted graphs [39, 51, 72]. Such approaches specialize in splitting identifiers containing abbreviations or acronyms.

TRIS [72] optimizes identifier splitting based on transformation rules and word frequencies. It first builds a acyclic directed graph to represent transformed dictionary words found in the source code, where a node represents a character and a path represents a transformation having a given cost. Second, it computes the cost of each transformation based on the frequency of its corresponding dictionary word and the weight of the applied transformation rule. Finally, it selects the path with minimum cost as the optimal split by using Dijkstra algorithm. Evaluation on dataset used by *GenTest* [103] suggests that the F-measure of *TRIS* is 96%.

Instead of defining transformation rules, *LINSEN* [51] optimizes identifier splitting by exploiting string matching algorithm Baeza-Yates and Perleberg (BYP) [21], which runs in linear time. It first builds a graph with each character of the identifier as the node, and the cost of matching dictionary word computed by *BYP* as the edge. Second, it searches the path with minimal cost using Dijkstra algorithm and returns the dictionary words matched in the path as the correct split. The dictionary that it exploits includes words from comments, source code, well-known abbreviations, and English dictionary. Case study on 4,741 identifiers extracted from two systems suggests that the F-measure of *LINSEN* is 80% and 95% on the two systems, compared to 70% and 93% of the approach proposed by Madani et al. [119]. Additionally, the authors compare *LINSEN* with *GenTest* [102] based on the dataset *GenTest* exploited. Results suggest that the accuracy of *LINSEN* ranges from 50% to 65%, compared to 35% and 58% of *GenTest*.

LIDS[7] [39] computes all possible splits and ranks the splits based on word frequencies in dictionaries. It first split each identifier into hard words by detecting division markers using regular expressions. Second, for each unknown hard word, it builds a graph with all possible soft words as nodes, and all possible sequences of path to concatenate the hard word as edges. Each edge is weighted, based on the word frequency and the weight of the dictionary where the word is found. The preferred dictionaries are, successively, a software-specific dictionary built automatically from documentation corpus, a general dictionary composed of programming terms and abbreviations, well-known and common acronyms, and an English dictionary provided by the *aspell*.[8] Finally, it chooses the soft words with the shortest path as the correct split by exploiting Dijkstra algorithm. The evaluation of the approach collects identifiers from two C projects. After removing duplicate identifiers and those with two or less characters, the dataset consist of 3,672 identifiers. The correct split of identifiers are manually checked by the authors. The result suggests that the F-measure of *LIDS* is 95%.

2.3 Expansion of Abbreviations

Expansion of abbreviations expands abbreviated non-dictionary words into dictionary words [87]. A trivial approach of expanding abbreviation is to look up its full words in a series of dictionaries in a certain order. However, it is challenging to map all abbreviations into their corresponding expansions based on predefined dictionaries because of the following reasons:

- The dictionaries created manually are only limited to abbreviations that the dictionary builders know.

[7]http://search.cpan.org/dist/Lingua-IdSplitter/.
[8]http://aspell.net.

- Expansion policies often depend on the context, e.g., abbreviation “PC” may be expanded into “Personal Computer” or “Patrol Car” in different contexts.
- New abbreviations come into use continually. Such newly created abbreviations cannot be expanded correctly based on predefined dictionaries.

Consequently, researchers propose various approaches to improve abbreviation expansion by exploiting different matching algorithms and different dictionaries. Table A.2 in the appendix highlights existing approaches proposed for abbreviation expansions.

Lawrie et al. [105], Hill et al. [77], and Jiang et al. [87] exploit heuristic rules to expand abbreviations. Lawrie et al. [105] expand abbreviations and acronyms by exploiting four dictionaries, i.e., a list of natural language words extracted from comments and identifiers in source code, a list of phrases extracted from comments and multi-words identifiers, a list of language stop words, and a natural language dictionary. When a word starts with the same letter and includes all characters in order as in the abbreviation, it is returned as the expansion. The approach ignores the cases when more than one expansions are retrieved. The qualitative evaluation of the approach involves 64 identifiers randomly selected from 158 programs (C,C++, and Java). The identifiers are manually inspected by three human evaluators who are given an identifier before and after expansion along with its function. Evaluation result suggests that the accuracy of the approach is at least 58%.

Hill et al. [77] look for abbreviation expansions by exploiting contextual information in the source code of method and program gradually. They classify abbreviations into different types (e.g., single-word abbreviations) and exploit different regular expressions to match the candidate long forms for each type based on a set of heuristics. If there are multiple candidate expansions within the same scope, the expansion with highest frequency is selected as the correct one. They implement the approach in *AMAP*[9]. The evaluation of *AMAP* involves 250 abbreviations randomly selected from five Java projects and expansions of such abbreviations are manually inspected by two human annotators who have no knowledge of the approach. Evaluation result suggests that the accuracy of *AMAP* is 59%.

Jiang et al. [87] propose to expand abbreviations involved in method parameters based on the lexical similarity between formal parameters and actual arguments. For a given abbreviation in actual arguments, the approach searches for it expansion from the corresponding formal (actual) parameter name, parameter type, and a given abbreviation dictionary by using heuristic rules. When an abbreviation has more than one candidate expansions, the approach selects the one more frequently expanded as the correct one. The evaluation of the approach involves 648 abbreviations in parameters sampled from nine Java projects. The evaluation data include no duplicate abbreviations and are manually expanded by three software developers who are not aware of the approach. The evaluation result suggests that the precision and recall of this approach is 95% and 65% respectively, compared to 26% of *LINSEN*.

LIDS [39], *TIDIER* [71, 118, 119], *TRIS* [72], *Normalize* [102], and *LINSEN* [51] focus on both identifier splitting and abbreviation expansion. Details of such approaches are illustrated in Section 2.2. *LIDS* [39] exploits general programming dictionary, known abbreviation dictionary, and custom dictionary generated using software natural language content to expand abbreviations. Experimental validation on dataset built from two open-source C projects suggests that the F-measure of *LIDS* in expanding abbreviations is around 91%. Madani et al. [119] looks for potential full terms by randomly choosing transformation rules and adapting Dynamic Time Warping (DTW) to match words in English dictionary. When there are multiple expansion candidates for an abbreviation, the approach randomly selects one. *TIDIER* [71, 118] reduces the number of transformation

[9]http://www.cis.udel.edu/hill/amp.

candidates by exploiting Hill Climbing algorithm, and exploits additional contextual dictionaries to look up. *TRIS* [72] converts abbreviation expansion problem into a minimum path matching problem in graph. It defines word transformation rules and assigns cost to each rule based on frequency and type of the transformation.

Normalize [102] exploits machine translation techniques to expand abbreviations and acronyms. It computes the probability of two words co-occurrence using statistical maximum coherence model [65], with a five word window on text extracted from source code, domain specific documentation, and the Google dataset distributed by the Linguistic Data Consortium [142]. It selects the word with the max probability of co-occurring as the expansion of an abbreviation. If more than one expansion for an abbreviation are potential, the approach makes no expansion. The evaluation of the approach involves 698 identifiers randomly selected from two C projects and manually inspected by four programmers. Evaluation result suggests that the accuracy of the approach is 66%.

LINSEN [51] exploits *ExpansionMatching* algorithm to discriminate expansion ambiguity based on high-level and domain specific information. Similar to the splitting policy (as illustrated in Section 2.2), *ExpansionMatching* algorithm adopts a graph representation for each abbreviation, and exploits the Dijkstra algorithm to look for the shortest path in the graph as the correct expansion. The cost of path is computed by applying soft BYP [21] algorithm to match dictionary words from source code, software system, computer science, and English dictionary. However, different from the splitting policy, *ExpansionMatching* algorithm selects matching words that only requires typical characters deletion operations. When ambiguities occur, the expansion within the more specific context is preferred. The F-measure of *LINSEN* in expanding abbreviations is 62%.

2.4 Part of Speech Tagging

The fundamental goal of part of speech tagging is to identify the grammatical category of component words in identifiers. Common POS taggers designed for natural language, e.g., *Stanford Log-linear POS Tagger* [24], *TreeTagger* [156], and *Minipar* [112], work well on highly structured sentences. Such taggers cannot work well on identifiers because the sequence of words extracted from identifiers rarely construct a grammatically-correct sentence [22]. Consequently, software specific POS taggers are proposed to tag POS for identifiers. Table A.3 in the appendix summarizes existing approaches of tagging POS for identifiers.

Liblit et al. [110], Falleri et al. [58], Binkley et al. [22], and Abebe and Tonella [6] apply templates as guidance for POS tagging. Liblit et al. [110] identify the POS of component words in identifiers based on predefined morphological naming conventions. It is the first work to develop a parsing tool for source code. Falleri et al. [58] apply template guidance for natural language POS tagger *TreeTagger* [156]. The evaluation of the approach involves 360 identifiers randomly selected form 24 Java projects and manually tagged POS by the authors. Result suggests that the accuracy of the approach range 83% to 96% for different types of identifiers. Abebe and Tonella [6] apply template guidance for the statistical POS tagger *Minipar* [112]. For each word extracted from an identifier, they first exploit *WordNet* to assign possible POS tags, second they constrain template rules that can be applied, and exploit *Minipar* to select POS from candidate template rules. Binkley et al. [22] identify POS for field names by applying templates as guidance for the Stanford Log-linear POS tagger [170]. This approach can only be applied to identifiers composed of dictionary words. The evaluation of the approach involves 1,500 field names randomly selected from 171 programs (C++, Java) and manually checked by student major in English. The evaluation dataset suggests that the accuracy of the approach is 88%.

Although template-based approaches improve the performance of traditional NLP taggers in tagging identifiers, they can only be applied in specific cases. POS taggers tailored for source code significantly outperform traditional NLP taggers in tagging identifiers [138]. Gupta et al. [74] develop a software specific POS tagger *POSSE* to tag for class names, method names, and attribute names in object-oriented programs. By hypothesizing that identifiers follow structural naming conventions, they identify possible POS tags for each component word using *WordNet* and chunk syntactic phrases for identifiers by exploiting structural rules described by Abebe and Tonella [6]. When more than one tagging rules can be applied to a word, they use Word Context Frequency Count (NVScore) to resolve ambiguity. The evaluation of the approach involves 210 identifiers (including class names, method names, and attribute names) randomly selected from 20 Java projects and 6 C++ projects. The dataset are annotated by one linguist and one experienced programmer. Evaluation result suggests that the accuracy of the approach is 91%.

srcNLP [12, 13, 133] assigns POS to component words of identifiers by mining stereotype data [9]. The approach first exploits *srcML* [44] to extract AST information (e.g., type of identifier) and gathers stereotype information about how an identifier is used based on static source code analysis. Second, it exploits heuristics to determine the POS of identifier words based on AST information and stereotype data. The evaluation of *srcNLP* involves verified identifiers from five open-source C/C++ projects. Result suggests that the accuracy of the approach is 91%.

Finally, Ye et al. [180] design a software-specific tagger *S-POS* by using a machine learning-based approach rather than empirical rules. They first extract 525 posts data from Stack Overflow [1], and build a software-specific POS tagset including the following features: word shapes, contextual tokens, and Stack Overflow tags. Second, they randomly deliver annotated data into training set and testing set, and train a Maximum Entropy Markov model (MEMM) to learn the POS. To evaluate the approach, the authors train MEMM model using 13,196 tokens and test model with 7,413 tokens extracted from Stack Overflow.[10] The dataset are annotated by three experienced programmers facilitated with Stanford Tagger. Evaluation result suggests that the accuracy of *S-POS* is 93%.

3 NAMING-CONVENTION–BASED RENAMINGS

Naming convention is a set of rules for programmers to guide them in naming software entities, especially the rules to form identifiers (which include choice of words but also "grammatical" rules). Naming software entities according to uniform naming convention improves readability and maintainability of software applications. Naming conventions are contextual [49]. Different programming languages and organizations define their own naming conventions. For example, Java language recommends naming entities following the camel case convention, whereas C programmers often concatenate words using underscores as separators.

Finding renaming opportunities based on naming conventions is different from the choice/quality of the naming conventions per se. It is not the naming convention per se but rather its violation that hints at problems. Theoretically, naming convention-based renamings must find first the contextual conventions adopted by the software and second identify identifiers violating such conventions as renaming opportunities. Table A.4 in the appendix summarizes existing naming-convention-based approaches.

3.1 Rule-Based Approaches

Rule-based approaches identify renaming opportunities by enforcing formal naming convention rules on the lexicon and grammatical structure of the identifiers. To standardize the lexicon used in identifiers, such approaches often build a standard dictionary first and identify identifiers containing non-dictionary words as renaming opportunities. To standardize the syntactic structure

[10] https://stackoverflow.com/.

of component words in identifiers, they identify identifiers with abnormal structure as renaming opportunities.

Caprile and Tonella [36, 37] propose a dictionary-based approach to standardize the lexicon of component words and the syntactic structure of function names. By extracting function names from ten C programs, the approach builds a standard dictionary to store standard words, a synonym dictionary to store the mapping from unknown non-dictionary words (e.g., acronyms, and abbreviations) into their standard words, and derives six grammar rules of constructing identifiers. To standardize the lexicon of identifiers, the approach identifies words not found in standard dictionary but listed in synonym dictionary as renaming opportunities. To standardize the syntax of identifiers, it detects identifiers violating extracted grammar rules as renaming opportunities.

Abebe et al. [5] constrain identifier constructing rules based on "lexicon bad smells" they defined. They develop *LBSDetectors* to identify the following lexical bad smells as renaming opportunities using POS tagger *Minipar* [112], *PaWs3*[11], and XML transformer *src2srcml*[12]:

(1) *Unusual Grammatical Structure*. Identifiers violating the following syntactic rules: the name of a class contains no nouns, the name of a class or an attribute contains verbs, and the name of a method does not start with a verb.
(2) *Terms Used as Both a Complete Identifier and Part of Another Identifier*. The lexicon construction problem is identified if one class name is used as part of the attribute or method name.
(3) *Inconsistent Identifiers Are Identified.* If two identifiers with the same entity type are located in the same container (entity) and one identifier is the part of the other.
(4) *Useless Type Indication*. Type information contained in the identifier is taken as redundant. Consequently, if attribute identifiers contain type information, then lexicon construction problem is identified.
(5) *Identifier Construction Rules Are the Predefined Naming Conventions Adopted by the System.* If an identifier does not follow such rules, then lexicon construction problem is identified.

To evaluate the performance of the approach, the authors randomly sample bad smells detected by *LBSDetectors* from two C++ projects. Manual investigation suggests that the precision of the approach ranges from 20% to 100% in identifying different kinds of bad smells (i.e., renaming opportunities).

Binkley et al. [22], Butler et al. [30, 31, 33], Kim et al. [96], and Corbo et al. [45] identify abnormal structure of identifiers by checking POS rules against naming conventions. Binkley et al. [22] constrain POS rules on field names. By investigating 145,000 field names from 171 programs (C++ and Java), they extract the following four POS rules for field names and identify those violating such rules as renaming opportunities: (1) non-Boolean field names should not contain a present tense verb; (2) field names should never be only a verb; (3) field names should never be only an adjective; and (4) Boolean field names should start with "is", "was", or "should".

Butler et al. [30, 31, 33] propose *Nominal* to constrain syntactic patterns of Java class names. Based on investigating 120,000 unique class names from 60 Java projects, they find that 90% of Java class names follow four grammatical forms: NN+, JJ+NN+, NN+JJ+NN+, and VBNN+. *Nominal* identifies names conflicting with such forms as renaming opportunities. Kim et al. [96] constrain different POS rules on different types of identifiers. For example, they identify class name *Restrict* as a renaming opportunity because the class name should be a noun phrase according to Java naming convention, whereas *Restrict* is a *verb* in *WordNet*. Corbo et al. [45] identify renaming opportunities

[11]http://ontoware.org/projects/paws/.
[12]http://www.sdml.info/projects/srcml/.

by checking the presence of prefix, separator, and the POS of the first word for different type of identifiers using *WordNet*. If one characteristic occurs over a given threshold for identifiers of same category, it is taken as a naming convention and used to identify bad names. They implement the approach in an Eclipse plug-in *Smart Formatter*.[13] An empirical study of *Smart Formatter* learns style on randomly mutated files from the same project. The evaluation result suggests that the performance varies with the threshold.

3.2 Learning-Based Approaches

Learning-based approaches propose mining emergent naming conventions based on statistical information from a codebase. They provide soft constraints on conventions, and the conventions mined can be flexibly applied to new projects. Reiss [148] extracts seven properties to feature naming conventions (e.g., whether the first character is upper or lower case) and trains different statistical learning techniques (e.g., decision tree, support vector machine (SVM), K-means, Maximum Entropy, and Bayes learner) to model such conventions. The approach computes all potential name changes based on applying learned models to arbitrary code and executes renamings using Eclipse refactoring capability.

Allamanis et al. [10], Lin et al. [111], and Suzuki et al. [166] exploit *n-gram* NLP techniques to identify renaming opportunities. *NATURALIZE*[14] [10] pioneers the application of *n-gram* statistical language model to look for irregularities. It identifies unnatural identifiers based on the probability distribution of all textual tokens linearly scanned in the code document using a moving window. Cross-validation on ten Java projects suggests that the accuracy of *NATURALIZE* is 94%.

Lin et al. [111] refine the lexical information exploited in the *n-gram* model on method level. They believe that syntactic symbols (e.g., punctuation, keywords, and comments) extracted from methods are noise for the language model. Consequently, they exclude such syntactic symbols when training the *n-gram*. The implementation of the approach, *LEAR*, can recommend local variables and parameters for methods. The evaluation of the approach involves manually inspecting 922 rename opportunities on five Java projects by seven developers and computer science students. The result suggests that the accuracy of *LEAR* in recommending meaningful names is 81%, compared to 47% of *NATURALIZE*.

Suzuki et al. [166] infer flexible common conventions by training *n-gram* model with sequences of words split from method names. The comprehensibility of each method name is computed based on the probability of its component words co-occurring in the *n-gram* model. Method names with low occurrence are likely to be incomprehensible and will be recommended proper new names with higher probabilities in the *n-gram* model. To evaluate the approach, the authors collect 1,000 Java projects from GitHub as a dataset to train a 3-gram model. By randomly selecting eight methods from the dataset and having them inspected by professional programmers, the evaluation result suggests that the precision and recall of the approach are 80% and 75%, respectively.

4 INCONSISTENCY-BASED RENAMINGS

Checking inconsistent identifiers in software is another critical technique to identify renaming opportunities. Identifiers convey domain concepts useful for program comprehension [76, 150]. They capture the application-specific knowledge that programmers possess when they write code [17]. Proper identifiers should not only concisely reflect the role of its entity, but also consistently represent the concept throughout the program. However, most programs suffer from inconsistent identifiers [100]. The challenges in preserving consistent names are explained as follows:

[13] http://www.rcost.unisannio.it/mdipenta/SmartFormatter.zip.
[14] groups.inf.ed.ac.uk/naturalize.

- First, naming conventions usually focus on syntactical aspects of naming entities, but provide little guidance to the semantics (meaning) of the identifiers [99]. Consequently, it is challenging for developers to name an identifier in such a way that matches the role of its corresponding entity well.
- Second, software evolution may lead to inconsistent identifiers. Developers may represent the entity in different ways during the long term of software development. For example, different identifiers are used to represent the same concept (i.e., synonymy) or an identifier is used to represent different concepts (i.e., polysemy).
- Third, aliases referring to the same instance and identical identifiers representing for different concepts within different scopes may also lead to inconsistent identifiers [49, 96, 147, 169].

4.1 Inconsistency between Identifiers

4.1.1 Synonyms and Polysemy-Based Approaches. Deissenboeck and Pizka [49], Deissenboeck and Ratiu [50], Ratiu and Deissenboeck [145], and Kim and Kim [96] propose to identify inconsistent identifiers based on mapping identifiers to real-world concepts. Deissenboeck and Pizka [49] enforce concise and consistent naming rules based on bijective mapping between identifiers and concepts. They derive rules of concise and consistent naming by analyzing the properties of identifiers, concepts, code, and the relationships among them by maintaining a software-specific identifier dictionary. The consistency rule constrains a bijective mapping between identifier and concept throughout the program, e.g., synonyms and homonyms lead to inconsistent identifiers. The conciseness rule constrains that the meaning of an identifier must exactly match the role of the entity. The implementation of the approach, *IDD*, enforces concise and consistent rules on constructing identifiers, and thus aids in consistent namings. For example, *IDD* identifies two identical identifiers with different types as renaming opportunities.

Deissenboeck and Ratiu [50], Ratiu and Deissenboeck [145] propose to identify semantic defects (e.g., synonymy and polysemy naming problems) by building a formal framework to map programs and real-world concepts. They first build a program layer to represent program entities, a lexical layer to represent identifiers, and a concept layer to represent concepts of the external knowledge. Second, they build a graph with identifers from lexical layer as nodes and relations from program layer as edges. Finally, they map the graph to an ontology of real-world concepts based on heuristic rules. Synonymy naming problem is identified when the same concept has different names in a program. Polysemy naming problem is identified when the same name represents different concepts.

Kim and Kim [96] develop *CodeAmigo*[15] to identify semantic and syntax inconsistency. To identify semantic inconsistency, the tool first extracts domain words and idiom words extracted from popular Java API documents, and stores such words together with the POSs of them in a code dictionary. Second, for each word extracted from identifiers in source code, it collects semantically similar words from the code dictionary using *WordNet* and computes similarity between them. Semantic inconsistencies (synonyms) are identified if the similarity is larger than the predefined threshold. For example, suppose *search* and *query* are all used as verb in a project and the verb *find* appears 20 times. By looking in the code dictionary, the tool finds that *search* and *query* are the first and 12th synonym words to *find* individually. According to the approach, the semantic similarity between *find* and *search* is 1-(1/20)=0.95, the semantic similarity between *find* and *query* is 1-(12/20)=0.4. If the predefined threshold is *0.6*, then *search* is identified as a synonym of *find*. Additionally, *CodeAmigo* identifies two words that have similar letter sequences as syntactic

[15]https://www.dropbox.com/s/i2nsqpx4k7prjt8/CodeAmigo_EclipsePlugin.zip.

inconsistency according to the following formula:

$$C(w_1, w_2) = \frac{max\{L(w_1), L(w_2)\}}{(|L(w_1) - L(w_2)| + 1) \times DIST(w_1, w_2)} \tag{2}$$

where $DIST(w_1, w_2)$ is the Levenshtein distance between word w_1 and w_2, $L(w)$ is word length. Syntactic inconsistencies are identified when the value of $C(w_1, w_2)$ is larger than a predefined threshold k. For example, identifiers $ServerCredential$ and $ServerCredental$ are identified as inconsistent names if the predefined inconsistency threshold is set to $k = 4$, because the Levenshtein distance between them is $DIST(Credential, Credental) = 1$ and the inconsistency value between them $C(Credential, Credental) = \frac{10}{(|10-9|+1)\times 1} = 5$ greater than the threshold. The qualitative evaluation of the approach checks inconsistent identifiers detected by *CodeAmigo* on seven Java projects. The result, which is validated manually by 16 developers, suggests that the precision and the recall of *CodeAmigo* is 85% and 84%, respectively.

Similar to mapping identifiers to concepts [49, 50, 145], Kamiya [89] develops *Vaci* to identify potential inconsistent names based on variation analysis and transitive closures. *Vaci* first detects pairs of context-sharing identifiers by parsing source code and generates transitive closures based on the contextual relationships between identifiers. Second, it classifies the transitive closures into different groups based on types of names and identifies distinct names and minority names in the transitive closures as renaming opportunities. It determines the minority names based on *n-gram* and clustering techniques.

Rather than using mappings between identifiers and concepts, Lawrie et al. [104] identify violation of synonym consistency and syntactic conciseness simply based on syntactic rules. A simple rule for consistency and conciseness is that concepts associated with two identifiers have no intersection. In other words, identifiers id_1 and id_2 violate syntactic rules if id_1 is a substring of id_2. Manual inspection of violations on two small programs suggest that the precision of the approach in detecting synonym violations and conciseness violations are 72% and 76%, respectively.

Thies and Roth [169] identify inconsistent names using variable assignment and static type information. They exploit a graph to represent variable assignments, in which each node represents a variable and each edge represents a variable assignment. In such graphs, edges that have no type change but hold a name change are potential inconsistent naming pairs. The approach recommends renaming less frequently used synonymous names and inaccurate names into more frequently used ones. An evaluation by hand on seven Java projects shows that 21 out of 32 warnings generated by the approach are due to inconsistent names. Consequently, the precision of the approach is 66%.

Abebe and Tonella [7] and Raychev et al. [147] recommend consistent names based on an identifier dependency graph. Abebe and Tonella [7] suggest semantic consistency in namings by extracting the ontology from the source code. The approach first splits the identifier into component words according to camel case and underscore, expands unknown abbreviations based on dictionary, and generates candidate sentences using *Minipar* [112] and syntax rules. Second, it builds an ontology by using the noun or noun phrase in the sentence as concepts and the dependency relations between concepts as ontological relations. Finally, it matches the newly typed word to the concepts in the ontology, and ranks all neighboring concepts in the ontology as candidate suggestions for the new identifier based on their relevance to the context of the new identifier. The evaluation of the approach involves 4,743 identifiers randomly sampled from six real projects (C++ and Java). Eighty percent of the recommended identifiers in case of term prefix are the real identifiers actually used in the programs.

Raychev et al. [147] propose *JSNice* to recommend consistent names based on *CRFs* model [101] and *MAP* inference [164]. The approach first transforms the input program into a dependency network graph, where each node represents a name, each edge represents a relationship between two names. Second, it computes the likelihood of replacing each unknown identifier with a known one based on the probability learned with *CRFs* from massive code bases. *CRFs* is a powerful undirected graph-based model for defining conditional probability distribution. Finally, the approach optimizes the nodes by maximizing the probability value of the network with Maximum *a Posteriori* (*MAP*) inference. The optimized nodes represent the recommended new names. To evaluate the approach, the authors download 10,517 JavaScript projects from GitHub as the training dataset and download 50 JavaScript projects with the most commits from BitBucket[16] as the testing dataset. The evaluation result suggests that 63.4% of identifiers predicted by the approach are exactly the names originally used in programs.

4.1.2 Clone-Based Approaches. Li et al. [109] propose frequency-based approach, *CP-Miner*, to identify inconsistent identifiers within copy-pasted code. The rationale of the approach is that if a developer changes most identifiers with the same name but forgets to change few of them in certain places, the unchanged identifiers have likely been forgotten. Consequently, *CP-Miner* exploits the frequency-based string mining algorithm *CloSpan* [179] to find copy-pasted code, map old name into corresponding new name within the copy-pasted code, and compute a value for the mapping. If the mapping value falls inside a defined threshold, *CP-Miner* identifies inconsistency between the old name and new name.

Jiang et al. [86] propose a context-based approach to identify inconsistent identifiers within clones. They first exploit the clone detection tool *Dekard* [85] to detect clone code. Second, they traverse AST trees of clones and compute inconsistency based on the contexts of clones, e.g., calculate the number of unique identifiers contained in a pair of clones. Finally, they filter out clone groups that do not strongly indicate bugs by ranking inconsistencies using heuristic algorithm and report remaining groups as naming bugs. Compared to *CP-Miner,* Li et al. [109], *Dekard* reports fewer false positive.

Verma et al. [175] propose a machine learning-based approach to identify semantic inconsistency (i.e., synonyms), syntactic inconsistency (i.e., similar word sequence), and POS inconsistency (i.e., violation of grammatical rules in identifiers) based on detected clones. The approach first builds a code repository to catalog identifiers from Java source code. Second, it exploits Ant Colony Optimization (ACO) [54] to scan identifiers in the repository and a back propagation neural network (NN-BPA) to detect inconsistency due to changes in source code. Evaluation on 12 Java projects suggests that the overall precision, recall, and F-measure of the approach are 84%, 75%, and 81%, respectively.

Higo and Kusumoto [75], Kusumoto and Higo [100], and Ray et al. [146] propose to identify inconsistencies across software revisions. Higo et al. [75, 100] identify inconsistent variables between software versions by extracting modification patterns (MP) from code repository. The approach first extracts *MPs* from a target system by analyzing past revisions using the Unix *diff* command, then quantifies *MPs* by calculating support and confidence, and detects inconsistencies in the latest source code based on selected *MPs*. The implementation of the approach, *MPANalyzer,*[17] detects inconsistencies caused by modifications of bug fixes, variables renamings, functionalities enhancing, and comments maintaining. The experimental evaluation manually checks 18 and 94 inconsistencies detected by the approach from two C projects. Result suggests that the precision on the two projects are 89% and 73%, respectively.

[16]http://bitbucket.org.
[17]https://github.com/YoshikiHigo/MPAnalyzer.

Ray et al. [146] design an algorithm called *SPA* to detect semantic inconsistencies (including inconsistent renamings) in ported code by statically analyzing control flow and data flow. By manually analyzing the commits containing "porting error" related keywords and corresponding source code, they find that errors often result from semantic inconsistency in ported code. Consequently, they first identifies the ported code by analyzing the reference patch code (set of added, deleted, or modified program statements) and target patch code. Second, it identifies the impact of the ported code by analyzing static control flow and dataflow. It constructs an isomorphic control dependence graph of the program, maps identifiers based on their syntactic similarity, and computes a confidence value for each mapping based on the times the mapping occurs. Finally, it reports inconsistent renaming if a map is not a one-to-one relation. By randomly selecting 132 samples from four open-source Java projects, manual inspection result shows that the precision and recall of *SPA* are 73% and 90% respectively.

4.1.3 Similarity-based Approaches. De Lucia et al. [47] exploit IR-based approach to identify renaming opportunities based on the textual similarity between high-level artifacts (e.g., requirement documents) and source code. They first exploit a *tf-idf* (term-by-document) matrix to represent words extracted from high-level artifacts and source code. Second, they transform the tf-idf space into concept-by-document vector space by exploiting the IR-based technique LSI (Latent Semantic Indexing) [48]. Finally, they compute the cosine similarity between the source code and artifacts in the vector space. If the similarity is lower than a threshold value, the approach identifies inconsistency between the identifier and the artifacts, and recommends to rename the identifier based on terms used in the high-level artifacts. They implement the approach in an Eclipse plug-in *COCONUT* to ensure the consistency between source code and artifacts. Two controlled experiments involve 36 university students and 4 Java projects are carried out to evaluate the usefulness of *COCONUT* in improving consistency between source code and high-level artifacts. Result suggests that more than 50% of identifier suggestions provided by *COCONUT* are accepted.

Liu et al. [114, 115] identify renaming opportunities based on the similarity between semantically related entities. They expand renamings opportunities based on the lexical similarity between the renamed entity and all its related entities (including sibling, inheritance, inclusion, and reference) [114]. The rationale of the approach is that if a developer badly names a software entity, he is likely to make the similar mistake in naming closely related entities. The approach works as follows: when a developer conducts a renaming operation $r1$, a monitor running in the background captures $r1$, an analyzer analyzes how $r1$ changed from the old name $oName$ to the new name $nName$ and generates a corresponding script of the transformation, a searcher searches all the entities that have the following relationships with $oName$: sibling, inheritance, inclusion, or reference. A recommender computes the similarity between the renamed entity $oName$ and each related entity $cName$ and recommends to do renaming $r2$ on the entity $rName$ whose name is most similar to $oName$. If the recommendation is accepted by the developer, this approach recommends a new name for other candidates based on $r1$ and $r2$. The recommendation triggered by $r1$ stops only if all candidates are renamed or the recommendation is rejected by the developer. Based on the assumption that identifiers follow camel case conventions, the approach is implemented in an Eclipse plug-in *RenameExpander.* The approach does not take synonyms into account when computing text similarity. The evaluation of the approach involves three participants manually checking recommendations generated by the approach on four open-source applications. The evaluation result shows that the precision of the approach in recommending renaming opportunities and in recommending new names are 82% and 93%, respectively.

Additionally, Liu et al. [115] identify renaming opportunities based on the similarity between *actual arguments* and *formal parameters.* For a given method invocation, the approach computes the lexical similarity between the argument *arg* and the parameter *par* and that between *par* and all of its alternative arguments. If *arg* is less similar (exceeds a certain threshold) to *par* than any alternative argument, the approach identifies *arg* as a renaming opportunity and recommends the alternative argument most similar to *par* as the new argument. To evaluate the effectiveness of the approach in anomaly detection, the authors manually identify known problems by analyzing the history of 14 programs and check whether the approach detects such problems. The evaluation result suggests that the approach identifies 144 renaming opportunities with the precision of 83%.

4.2 Inconsistency between Identifiers and Entities

The approaches mentioned in the previous section take only identifiers as a main feature to detect inconsistencies, but without considering the inconsistencies between the implementation of an entity and the name of the entity. It should be straightforward to understand the role of entities having descriptive names even without reading their implementation details, whereas it is difficult to understand the role of entities with misleading names. Consequently, some researchers propose to rename the entity to reflect its implementation when the name of an entity is inconsistent to the implementation of the entity.

Høst and Østvold [78, 79, 80], Karlsen et al. [91], and Singer and Kirkham [160] identify inconsistency between entity name and entity implementation based on micro-patterns [67]. Micro-patterns are implementation-oriented low-level patterns. Høst et al. [78] link the meaning of the verb in the method name to the semantic attributes of the method body by analyzing the micro-patterns found in the compiled byte-code instructions and build a phrase book[18] [79] to capture the grammatical structure of method name and the implementation rule (represented by a semantic profile) of the method body. For example, according to the implementation rules, a method containing *find* in the name should read local variables, contain loops, check types, and have a return value. The implementation of the approach (i.e., *Lancelot* [91]) can accurately point out inappropriate verb of method name and recommend more appropriate verbs. *Lancelot* filters out relevant names violating implementation rules and sorts the candidate phrase list according to the rank of semantic profile or the semantic distance between old name and candidate new phrase. The evaluation of the approach involves 50 reported bugs randomly chosen from 100 Java applications. Manual inspection suggests that the precision of the approach is 70%. Singer and Kirkam [160] identify inconsistencies between the suffix of class names and micro-patterns of class implementation. By analyzing large numbers of real-world Java programs, they infer micro-patterns of the implementation from the suffixes of class names and develop a tool called *non-interactive checker* to detect violations of such patterns. Evaluation on nine Java projects suggests that the accuracy of *non-interactive checker* is 75%.

Similar to inferring micro-patterns from entity implementation, Kashiwabara et al. [92, 93] propose to rename the verb contained in method names for consistency by mining association rules [82] from method implementation. They assume that the functionality of method can be characterized by method invocation and field access statements in the method body. Consequently, they first extract the verb in the method name, the return type of the method signature, and invoked methods in the method body into a transaction. Second, they mine naming association rules in the form (*antecedent, consequent, confident, support*) based on extracted information. For example, if 90 out of 100 methods whose names start with "get" call a method named *getItem* in the implementation, the extracted naming association rule would be ({*call:getItem*}, *get, 0.9, 90*). Finally, the approach recommends verb for the method name based on all applicable rules. If more than one

[18] http://phrasebook.nr.no.

rule recommends different verbs, it presents a list to developer and recommends the one with the highest confidence. The evaluation of the approach involves training naming association rules on 445 open-source Java projects and testing such rules on six open-source Java projects. The evaluation result suggests that 60% of the methods are recommended the correct verb by the approach in the top 10 candidates.

Different from detecting inconsistency in source code, Arnaoudova et al. [18, 19] propose to identify inconsistency among the software artifacts based on software *Linguistic Antipatterns*. They identify the following six kinds of *Linguistic Antipatterns* from methods as renaming opportunities: (1) Do more than it says, e.g., a method whose name starts with *is* returns an object; (2) Say more than it does, e.g., a method whose name starts with *get* returns a bBoolean value; (3) Do the opposite, e.g., a method named *getValidateEmails* returns a list named *InValidEmails*; (4) Contain more than it says, e.g., attribute ending with a singular noun form like *student* has a type of *List*; (5) Say more than it contains, e.g., attribute ending with plural noun form like *grades* has a type of *boolean*; and (6) Contain the opposite, e.g., a method with name *remove* has comments *add a new student to the list*. Based on structural and textual analysis of source code, they implement the approach in *LAPD*. *LAPD* first extracts method names, return types, parameter names and types, control flow, and other information from the source code using *srcML* [44]. Second, it splits identifiers into component words using camel case and underscore heuristics, and performs POS analysis using Stanford POS tagger. Finally, it identifies inconsistency against defined Linguistic Antipatterns using corresponding heuristic algorithms. For example, *LAPD* identifies the method whose name starts with "is" but the return type is not Boolean as a renaming opportunity. The qualitative evaluation of the approach samples inconsistencies detected by the approach from four Java projects and manually checks them. The evaluation results suggest that the precision of *LAPD* is 72%.

Instead of manually coding rules for the names and implementation of entities, Yu et al. [181], Allamanis et al. [11], and Liu and Kim [116] propose machine learning-based approaches to learn inconsistency between entity names and entity implementation. Yu et al. [181] train a classifier to recommend method names based on SVM [26] technique. The approach models methods and variables by featuring the function and structure of Java methods and the context of variables. For example, it captures signature features (e.g., parameter types), content features (e.g., the number of objects created), and sub-verbs (e.g., verbs of methods invoked) for methods. Second, it builds a vector space model to represent each feature. Finally, it feeds feature vectors and the POS of the verb and the object of method name into an *SVM-based* classifier, which learns proper verb and proper object for method name. The approach predicts the verb of an unnamed method and whether a variable is or not the target of the method. By randomly selecting 60% methods from three Java projects as training set and 40% as testing set, evaluation result suggests that the accuracy of the approach in predicting verb and target is 70% and 90%, respectively.

Allamanis et al. [11] exploit log-bilinear neural network to suggest method and class names. By featuring local context (i.e., tokens from method body) and global context (i.e., a set of feature functions), the approach assigns semantically similar words with similar location in a high-dimensional vector space. It highlights the name whose vector is not similar to that of its contexts and suggests a new name which is most similar to the method body in the vector space. It also exploits the internal structure of identifiers to model a sub-token context, which makes it possible to suggest neologisms that never appear in the training set. The qualitative evaluation of the approach splits files from top 20 active Java GitHub projects into 70% training set and 30% testing set. Evaluation result suggests that the approach predicts neologisms with F-measure of 50%.

Liu and Kim [116] exploit *Paragraph Vector* [107] and Convolutional Neural Network (CNN) [121] to identify inconsistent method names and suggest better ones. The approach first extracts method names, and AST node types and tokens of method bodies by parsing AST of methods.

Second, it exploits *Paragraph Vector* to embed method names, and exploit *Word2Vector* [126] to embed method body and *CNN* to decrease dimensions. Finally, it identifies whether a method name is consistent to its body by comparing them in distinct vector spaces. The empirical validation of the approach trains the CNN model with 430 open-source Java projects and tests the model with 2,805 distinct methods having buggy names and fixed one in two versions. The evaluation result suggests that the F-measure of the approach in identifying inconsistent method names is 68%.

Summarization of inconsistency-based renamings is shown in Table A.5 in the appendix. In summary, existing approaches identify renaming opportunities based on the inconsistency between identifiers and the inconsistency between the name of entity and the implementation of entity. Furthermore, inconsistency between identifiers are classified into synonyms-based approaches, clone-based approaches, and similarity-based approaches.

5 EXECUTION OF RENAMINGS

Theoretically, in non-trivial programs, it is impossible for a tool to guarantee behaviour preservation just for the fact that behaviour is ill defined at the best of times but also because of branches, the halting problem, etc. In practice, complex implementation of renaming tools may contain bugs, which makes renaming even more risky. Behaviour preservation is mostly ensured by checking a set of preconditions [139, 151]. However, it is difficult to define sufficient preconditions because it is complex to define name lookup rules and new language features require additional preconditions [152]. As a result, even the up-to-date IDEs (e.g., Eclipse) may break name binding preservation [61].

The main task of executing renamings is preserving name bindings, i.e., to actually rename the identified identifier and propagate renamings in such a way that the program will compile and run after renamings. However, it is challenging to preserve name bindings because of the following reasons:

- It is tedious to manually rename all references to the identifier one-by-one because there are often many occurrences of them. Forgetting to rename any one of them may introduce potential bugs.
- It is error prone to syntactically search and replace identifiers. Search-and-replace-based approaches traverse source code as plain text or abstract syntax tree, but takes no semantics into consideration. As a result, such approach may result in wrongly renaming unrelated entities with the same name.
- Any new names introduced in renamings may introduce name conflicts, change name binding, and produce syntax errors. To preserve name binding, renamings must take into account all semantic rules (e.g., namespace, scope, and visibility) implemented by the compiler.
- When the program involves elements written in different languages, name binding analysis must take into account multi-language name binding.

To preserve name bindings as much as possible, existing approaches: (1) specify preconditions to ensure that the transformed program is compilable, executable, and that its behaviour is preserved as much as possible and (2) search for related references to the identified identifier and rename them consistently. Table A.6 in the appendix summarizes existing approaches of applying renamings.

5.1 Precondition-Based Approaches

Checking preconditions before conducting renamings is one way to ensure that renamings will produce a program that compiles and executes the same behavior as the original program [140].

However, it is difficult to derive concise preconditions because theoretically such preconditions should take into account the same semantic rules that the compiler implements. For example, preconditions for renamings in Java should take the following features into consideration: namespaces, name shadowing, name visibility, language words, reflection, naming conflicts, and so on [141]. Too weak preconditions may lead to binding changes after renamings, whereas too strong preconditions prevent certain kind of renamings. Meanwhile, preconditions may become outdated when new features are brought with language evolution [152]. Even refactoring frameworks in current IDEs cannot guarantee naming bindings because of insufficient preconditions [163]. Consequently, researchers proposed different approaches to specify preconditions to ensure that naming bindings are preserved as much as possible.

Cohen [43] presents a prototype for renaming global variables in *C* programs based on a semantics formalized in *Coq*. The approach is the first work in formally proving behavior preservation for an industrial language. It handles situations of name shadowing and gives a sufficient precondition for behavior preservation by verifying the logic of AST transformation.

Burgula and Reddy [27] supports to rename identifiers accessed by external code and designs a precondition-based renaming tool *Go Doctor* for the statically typed programming language *Go*. *Go Doctor* checks the input validity and transformation validity by parsing ASTs of all file dependencies using Go/Loader [3]. The input validity ensures that the new name is legal. The transformation validity ensures no name conflicts in the current scope, child scope, and parent scope. For example, when renaming method, it checks name conflicts among methods with same receiver type. *Go Doctor* aborts the renamings when they involve name shadowing.

Overbey and Johnson [140] and Overbey et al. [141] present a language-independent precondition checker to guarantee input validity, program compilability, and behavior preservation after renamings. They first build a graph to model the name binding of the original source code, simulate the renaming transformation, and model the name binding of the transformed program. Second, they construct a reusable, generic differential precondition checker to look for differences between the name binding of the original program and that of the transformed program, and perform behavior preservation analysis to guarantee that the differences are expected. Renaming operations are applied to the code only when the precondition checker guarantees that each identifier binds to the same entity before and after the transformations.

5.2 Reference-based Approaches

As discussed in Section 5.1, it is challenging to extract proper preconditions to guarantee name bindings, even based on analyses on code base in large scale, because theoretically it is impossible to cover all problematic cases without aborting useful renamings [152]. Consequently, instead of checking preconditions, some researchers choose to execute renamings based on identifier references, i.e., find and update all references to the renamed identifier consistently.

Static-Language. Jablonski and Hou [83] propose *CReN* to rename all instances of the same identifier consistently within copy-pasted clones in Java program. When users copy and paste code, *CReN* detects such operation and clusters identifiers into different groups based on their relationships in the ASTs. Identifiers sharing the same binding are classified into same group. When an identifier in the pasted code is edited, *CReN* will automatically rename identifiers from the same group consistently. Jablonski and Hou [84] extend *CReN* into *LexId* to infer common patterns from different identifier instances by tracking clones and dividing those having common substrings into same groups. Consequently, *LexId* can identify the same part of different identifiers that should be renamed consistently and rename them automatically.

Guo et al. [73], Schäfer et al. [152], and De Jonge et al. [46] bind references to entity declaration by creating globally unique identifiers. Guo et al. [73] assign a global unique identifier to each entity declaration based on declaration context in Java program and attach the unique identifier to entity references using XML tags. Schäfer et al. [152] guarantee name binding by creating symbolic names and inverted lookup rules. By using the *AST* as a symbolic table, they bind each identifier to its corresponding declaration by a name lookup function. The lookup function specifies a unique tree location for each identifier and generates a symbolic name for the identifier. The approach also binds each entity declaration to all of its references by an inverted name lookup function. When developers perform renamings, the approach tries to create symbolic names to guarantee name bindings when name conflicts may occur after the renaming transformation. If it cannot generate a unique symbolic name, the approach terminates the renaming operation and roll back all changes made. De Jonge and Visser [46] preserve static name binding by creating qualified names. Similar to name binding analysis used in the compiler, they generate global unique reference names for all identifiers in the source code. Two identifiers are annotated with the same reference name only if they bind to the same declaration. Name bindings are preserved only if the original reference names are equal to the new names after renamings. When name bindings are violated after renamings, the approach restores name bindings by looking up and creating qualified names in a general namespace. If no qualified names can be generated, the renaming transformation will be rejected.

Dynamic-Language. Compared to static typed languages, it is difficult to automate renamings for dynamic programming languages, like JavaScript, because no static type or declaration information is available. Few IDEs, except WebStorm,[19] provide support for renaming dynamic language programs. Feldthaus et al. [60] and Feldthaus and Møller [61] exploit static point-to information to approximate name binding analysis in JavaScript renamings. They divide all occurrences of an object property *x* into groups by analyzing how objects are used and a set of constraints generated by a type inference system. All *x* in the same group are renamed simultaneously when developers rename a certain occurrence of *x* in the group. Compared to the search-and-replace technique used in current JavaScript IDEs, the proposed approach can save 57% of manual effort in conducting renamings. The implementation of the approach, *JSRefactor*,[20] rejects the renaming entities whose parent object may be accessed reflectively when dynamic property expressions are involved.

Multi-Language. Kempf et al. [95], Chen and Johnson [41], Schink and Kuhlemann [154], Schink et al. [155], Nguyen et al. [136], and Mayer and Schroeder [124] propose different approaches of renamings for Multi-Language Software Applications (MLSAs) [113]. Renamings in MLSAs involve analyzing the semantics of the individual languages and multi-language bindings. Kempf et al. [95] rename Groovy[21] language code embedded in Java programs by searching references to them with a Java search engine. It is challenging to rename Groovy elements in Java programs based on refactoring provided by Eclipse JDT because references to Groovy entities are visible only at binary level and renaming elements at binary level is not supported. Consequently, the approach proposes to find Groovy elements first using a Java search engine and then generates edits to rename the found references accordingly.

Chen and Johnson [41] propose automated renamings across three popular Java frameworks (i.e., Struts, Hibernate, and Spring) by studying the interactions between Java files and XML configuration files. When the approach detects an initial renaming performed on a Java file by the user, it goes through all the XML files in the project and filters out relevant XML files. If the approach finds that a file refers to the newly renamed identifier, it creates necessary change to the references in the file and triggers additional refactorings to keep program consistent.

[19]http://www.jetbrains.com/webstorm.
[20]https://www.brics.dk/jsrefactor/plugin.html.
[21]https://groovy-lang.org/.

Schink and Kuhlemann [154] and Schink et al. [155] rename related artifacts of multi-language Hibernate applications consistently using the object-relational mapping (ORM). In Hibernate applications, Java elements interact with the corresponding tables in the database schema based on the *ORM* created by Hibernate. When developers rename Java elements, the interaction between them may break and thus results in runtime exceptions. To restore the mapping between the renamed entity and its corresponding database schema, the approach proposes to add Java annotations to the renamed methods. For example, when developers rename method *getSalary* to *getMonthlySalary*, the approach automatically adds an annotation *@Column*(*name*=*"salary"*) to the method so that the method with the new name can map to the column *salary* of the database schema.

Mayer and Schroeder [124] present a prototype for identifying and renaming multi-language artifacts in Java software systems based on the binding logic of artifacts. They create a dedicated binding resolution algorithm to resolve information about which properties contribute to the name binding and how such names interrelate. If developers rename one artifact, the corresponding renaming edits propagate to all artifacts bounded through the multi-language routine. The actual renamings is performed by language-specific refactoring routines when related artifacts come from different languages.

Nguyen et al. [136] present *BabelRef* to automatically rename PHP-based Web applications consistently. *BabelRef* first symbolically executes each *PHP* page and represents all generated client pages with *D-models* tree structure [135]. Second, it identifies cross-language entities and their references by discovering the semantics of leaf nodes in the *D-model*. When users rename a certain entity, *BabelRef* symbolically executes the *PHP* program, identifies related entities in the background, and regenerates the entity and reference lists automatically.

Language-independent. Language-Independent. Verbaere et al. [174] propose the idea of designing the script language *JunGL* to describe *rename* (and other) refactoring. Built upon a general-purpose functional programming language ML (Meta Language) [171] and Datalog [40], *JunGL* represents a software as a graph by adding additional semantic and syntactic information as "Lazy Edges" to ASTs. The path in the graph represents dataflow properties and thus can be used to lookup name references in renamings. The approach builds upon rule-set of languages and, as a result, is language independent.

Ge et al. [66] present *BeneFactor* to detect ongoing field name renamings based on a set of workflow patterns extracted from manual refactoring. By studying the videos of renaming operations, they found that participants perform renamings according to two patterns: (1) Update the field's name and iteratively update all references names and (2) Use the "find and replace" functionality to automatically replace all occurrences of the field's name. *BeneFactor* models such workflow patterns using finite-state machines (FSMs) and detects an ongoing renamings by matching operations performed by the developer against extracted workflow pattern. If the matching confidence exceeds a predefined threshold, it detects manual renamings and complete renamings automatically.

6 DETECTION OF RENAMINGS

In previous sections, we summarized the approaches involved in the process of renamings, i.e., identification of renaming opportunities, recommendation for proper new names, and execution of actual renamings. All of these approaches are related with the process of automatically or semi-automatically improve the readability and the comprehension of the code.

In this section, we summarize approaches on detection of renamings. Detection of applied renamings is not directly related to the traditional renaming process. We present this section (detection of renamings) because of the following reasons. First, some renaming approaches, e.g., RenameExpander [114], are based on the detection of renamings. Second, detecting renaming histories significantly facilitates the research on renamings by providing a large number of real-world renaming occurrences. Finally, detecting renaming histories facilitate the comprehension of software evolution. Although existing IDEs provide renaming history information, such information is not sufficient. On the one hand, renaming information recorded in IDEs are not accessible to researchers but only available to developers on the client side. On the other hand, IDEs only record renamings performed using *rename refactoring* menu, whereas do not provide the recorded refactoring information conducted manually or based on other tools. However, according to Murphy's report [130], 32% of refactoring are performed by hand rather than using any refactoring tools. Consequently, renaming information recorded in IDEs are not sufficient. Researchers propose different approaches to detect, classify, and analyze conducted renamings.

Malpohl et al. [120] exploit a suite of differencing and merging algorithms called *renaming detector* to detect renamed variables. The approach first catalogs all identifier declarations and links them to the corresponding references by exploiting differencing and symbol analysis. Second, it uses a heuristic-based algorithm to match declarations in two versions and detects renamings based on the types of variables and the similarity of the reference chains of each declaration computed by the *Jess* expert system [64]. The approach is insensitive to code formatting and moved code blocks. It treats comments and programs differently and can adapt to data description languages (e.g., XML). The evaluation of the approach on 77 Java files correctly detects 40 out of 47 real world renamings. Consequently, the precision and recall of the approach in detecting renamings are 100% and 85%, respectively.

Kim et al. [97] detect renamed functions across revisions by computing the similarity between functions based on the following eight weighted similarity metrics: (1) the similarity between names of the function pairs; (2) the similarity of incoming call set; (3) the similarity of outgoing call set; (4) the similarity of signature pattern; (5) the similarity of function body computed by text diff; (6) five complexity metrics (e.g., cyclomatic complexity); (7) the similarity computed by plagiarism detector MOSS [8]; and, (8) the similarity computed by clone detector CCFinder [90]. If the similarity of two functions is greater than a predefined threshold, the approach identifies renamed functions. To evaluate the approach, the authors samples 20% revision history from two open-source C projects and ask ten human judges to manually identify renamed entities. Evaluation result suggests that the accuracy of the approach in detecting renamings is 91%.

Neamtiu et al. [131], Fluri et al. [63], and Kawrykow and Robillard [94] detect renamings based on matching identifier declarations and references in ASTs. Neamtiu et al. [131] detect renamed global variables, functions, and types in C programs based on bijection AST matching. Fluri et al. [63] propose a tree-based differencing algorithm *Changedistiller* to detect renamed methods and fields in Java programs. The algorithm compares the leaf nodes in the trees across versions of Java files, and computes the similarity between them using *bigram string similarity* and *subtree similarity*. To evaluate the approach, the authors conduct case studies on 1,064 manually checked methods changes from three different open-source projects revisions. Evaluation result suggests that the precision of the approach is 66%. Kawrykow and Robillard [94] extend *CHANGEDISTILLER* as *DIFFCAT* to detect class, local variable, and field renamings.

Arnaoudova et al. [20], Eshkevari [56], and Eshkevari et al. [57] propose *REPENT* to detect and classify renamings based on differencing and data flow analysis. *REPENT* first maps lines of code and declarations between revisions to identify possible renaming pairs by using line-based difference tools, e.g., Unix *diff*. Second, it filters out false positives from candidates renaming entities by applying *def-use* analysis. Finally, it classifies the detected renamings into the following four orthogonal dimensions using *WordNet* [127] and the Stanford POS Tagger [24]: (1) entity kind; (2) semantic change; (3) string distance between the terms; and (4) grammatical change. They find that: (1) synonyms, homonyms, hypernyms, hyponyms, and antonyms highlight inconsistent and inconcise identifiers and thus induce renamings; (2) renamings of methods and field identifiers occur more frequently because developers need to reflect behaviors and properties changes; (3) the grammar forms of terms in the renamings do not change often; and (4) 80% of the renamed terms add meaning to or remove meaning from original identifiers. Twenty percent of the renamed terms generalize, specialize, remain, or have opposite or unrelated meaning to the original terms. Eighty percent of the renamings have high textual similarities between the original and the renamed terms. For the evaluation, Eshkevari et al. [57] manually analyze a sample of 330 classified renamings from five open-source Java projects in different domains. The evaluation result shows that the overall precision and recall of *REPENT* in classifying renamings are 88% and 92%, respectively.

Based on the taxonomy created by *REPENT*, Peruma et al. [143] further categorize renamings and analyze why developers rename identifiers based on commit messages. They first detect renaming operations using *RefactoringMiner* [159]. Second, they analyze the semantics and parts-of-speech of identifiers using python *NLTK* [117]. Compared to *WordNet* used by *REPENT*, *NLTK* uses multiple stemming and lemmatization techniques to find relationships between two identifiers. They find that: (1) most (45%) of the renamings narrow down the meaning of identifiers; (2) a majority (59%) of the renamings change only one term of identifiers; and (3) 70% of changed terms in renamings demonstrate grammar (i.e., part-of-speech) change. They also analyze commit messages to identify the reasons for renamings by exploiting a topic modeling algorithm to infer the main topic of documents. Results suggest that: (1) developers perform renamings in *test*, *rename*, and *fix* related commits often and (2) when renamings narrow, broaden, or add meaning to identifiers, words *add* and *ad* (stemming from *add*) often appear in commit messages.

Besides the approaches specifically designed to detect renamings, some generic refactoring detection approaches [53, 68, 159, 176–178] can detect renamings as well. Weissgerber and Diehl [176] detect and rank renamed methods and classes by syntactic and signature analyses. They first look for changed entities between versions with a syntactical analysis using regular expressions. Second, they define the criteria of refactoring candidates based on signature analysis. Finally, they rank such candidates to refine real refactorings using clone detection to check behavior preservation. Godfrey and Zou [68] detect renamed entities based on origin analysis [172]. Origin analysis uses syntactic and semantic analyses to decide whether a program entity is newly introduced, renamed, moved, or otherwise a changed version of an original entity. They provide a list of potential origins of a target entity based on different matching techniques, e.g., name matching, declaration matching, metrics matching, and call relation matching.

Xing and Stroulia [177, 178] detect and classify refactorings on a design-level based on the differencing algorithm *UMLDiff*. Given two class diagrams, *UMLDiff* produces a XML file showing the design difference between them. By querying the XML file, it can detect rename (and other) refactorings.Van Rysselberghe and Demeyer [173] propose "software palaeontology" heuristics to reconstruct evolution processes by analyzing the differences between different software releases. They detect rename refactorings based on clone detection and *dotplots* [42]. *RefactoringMiner* [159] detects refactoring using differencing object-oriented models. They find that refactorings are mainly driven by requirements other than code smells.

Dig et al. [52, 53] develop *RefactoringCrawler* to detect refactoring candidates based on Shingles encoding [25] and determine whether the semantic relationships between two candidate entities represent refactoring based on the reference-graph between entities. As a result, *RefactoringCrawler* cannot detect refactorings if API references are not available. Taneja et al. [168] develop *RefacLib* to detect API refactoring based on Shingles encoding as well. Instead of depending on API references, *RefacLib* exploits a set of heuristic rules to analyze program pairs.

Alves et al. [14] propose to detect fault (introduced after refactoring) by prioritizing regression test cases. They first exploit refactoring fault models (RFMs) [127, 132] to detect methods affected by *rename method*, and then reorder existing test suites for such methods to foster early detection of the fault introduced after renamings.

To summarize the detecting renamings surveyed in this section, Table A.7 in the appendix classifies all existing approaches. The table summarizes to what types of entities each approach applies and what kinds of information and techniques each approach considers. Furthermore, the table presents the performance of each approach and the evaluation dataset. We find that all approaches/tools for detecting renamings are not publicly available.

7 CONCLUSION AND FUTURE WORK

Identifiers are important sources of information for program comprehension and software maintenance. Proper identifiers act as bridges between the concepts in the developers' mind and the entities in software. However, for reasons like time schedule, company culture, personal preferences, and the like, entities in millions of software are often improperly named.

Renaming is one of the most popular refactorings, restructuring improperly named identifiers. The challenge in automating renamings lies in: (1) it involves complicated NLP analysis in mining the information embedded in identifiers accurately, which is often influenced by the culture of companies and developers' preference and (2) it involves complicated semantic analyses to gain the meaning of entities.

We may identify renaming opportunities and recommend new names by comparing the syntax of component words in identifiers against naming conventions and by mapping the semantic of identifiers to the role of their corresponding entities. When improper names are identified, researchers propose different approaches to transform them into new names and automatically update all references to them consistently so as to preserve name bindings. Additionally, researchers detect and analyze conducted renamings to better understand why and how identifiers evolve, which may help to validate the performance of proposed renaming approaches, and thus improve program comprehension.

To reveal the state-of-the-art and to facilitate comprehension of related work, this article provides a survey on renamings of software entities. We analyze existing approaches on renamings, compare them in a structured way, and present the surveyed papers according to the problems they address. To the best of our knowledge, it is the first survey work on this area.

Some of the key conclusions are presented as follows:

(1) Renamings are gaining more and more attention in academic community. Compared to the number of papers published in early years, the number of papers published in the last 10 years increases by an order of magnitude. The increasing number of papers suggests that research in this area is booming.
(2) Existing research covers all key aspects of renamings. Of the surveyed papers, 29% (=32/109) focus on the preprocessing of identifiers, 36% (=39/109) focus on the identification of rename opportunities, 18% (=20/109) focus on the execution of renamings, and 17% (=18/109) focus on the detection and analyses of renamings.
(3) Identification of renaming opportunities is often integrated with the recommendation of new names. By analyzing improper names, i.e., when the names are improper, we can infer how to revise (rename) the names to change them from *improper* into *proper*. Among the 39 papers on identification of renaming opportunities, 79% (=31/39) of them recommend new names as well, whereas only 21% (=8/39) of them do not recommend new names.

(4) We lack a public and comprehensive dataset for evaluation. Different approaches are evaluated on different datasets that are often not publicly available. Among the 109 papers covered by the survey, 88% (=96/109) conduct evaluation. Among the 96 evaluated approaches, only 13% (=12/96) make their dataset publicly available and 5% (=5/96) are evaluated on publicly available datasets that have been employed for evaluation by other approaches.

(5) Most of the implementations of the related approaches are not publicly available. Among the 109 approaches covered by the survey, only 10% (=11/109) make the corresponding implementations (source code or executable software systems) publicly available. Failing to publish the implementations makes it difficult for other researchers/readers to repeat experiments or to compare related approaches. Failing to publish the implementations also results in a significantly smaller chance to be employed by the industry.

Besides building a public and comprehensive data set for evaluation of different renaming approaches/tools, we list here some potential research directions in this area:

(1) *Comprehensive Dataset.* Exisiting renaming approaches are evaluated on different datasets that are often not publicly available. It could be interesting and highly valuable to build and publish a comprehensive dataset for evaluation of renaming approaches.

(2) *Renaming Execution and Detection.* From Figure 4, we observe an imbalance between the three types of approaches. It would be valuable to fill the gap in the works of renaming execution and detection.

(3) *Usefulness of Advanced Identifier Splitting Approaches.* As suggested by Tables A.1–A.3, preprocessing of identifiers is one of the hot topics in renamings. For example, researchers proposed a large number of advanced approaches to split identifiers into a sequence of tokens [29, 51, 55, 59]. Such approaches are based on the assumption that an essential part of the identifiers do not follow well-known naming conventions, like the camel case naming convention [71] and such identifiers should be split by advanced splitting approaches. However, by analyzing papers on renamings, we find that such advanced splitting approaches are rarely employed by renaming tools/approaches. In contrast, most of such tools/approaches [7, 18, 19, 114, 115] employ simple and straightforward heuristics: splitting identifiers according to capital letters and underscore. It would be interesting to investigate the reasons for using such simple heuristics rather than more advanced approaches.

(4) *Preserving Name Bindings.* The main task of executing renamings is to preserve name bindings. Name bindings are often ensured by checking a set of preconditions or searching all references to identifier declarations [27, 140, 141]. However, it is challenging to define sufficient and accurate preconditions [46, 73]. Too weak preconditions may result in name bindings change after renamings, whereas too strong preconditions prevent legitimate renamings [152]. It is also impossible to find all references to identifier declarations [60, 61]. Name shadowing, polymorphism, inheritance, dynamic type, third-party library, and multi-languages systems make it challenging to guarantee name bindings. The evolution of languages also imposes the challenge of adapting existing name binding approaches to new features of languages (e.g., *reflection* in Java). It would be interesting and valuable to investigate the preservation of name bindings based on language features.

(5) *Representation of Identifiers.* Deep learning techniques have been applied in the process of renamings, e.g., preprocessing identifiers, identifying renaming opportunities, and so on [11, 98, 116, 181]. The performance of deep-learning-based renamings depends on vector representation of identifiers [116]. Various vectorization approaches (e.g., *Word2Vector*

[126], *Paragraph2Vector* [107]) have been applied to identifiers (as well as their contexts) [11, 116]. However, little is known about the underlying rationale for different choices. It would be valuable to investigate/compare the performance of different representation techniques in renamings, and to propose new techniques if needed.

(6) *Identifier Obfuscation.* Renaming is to improve program comprehension. However, for reasons like software security or intellectual property protection, developers may deliberately transform meaningful identifier into short, meaningless, or arbitrary obfuscated one. Such obfuscated identifiers hinder program comprehension, hide the role of software entities, and thus make it difficult to conduct manual and automated analysis on source code. It would be interesting to investigate how techniques used in renamings can be used to reduce the readability of the code.

REFERENCES

[1] 2017. https://stackoverflow.com/.
[2] 2019. https://www.unisonweb.org/docs/tour.
[3] 2019. https://godoc.org/golang.org/x/tools/go/loader.
[4] 2020. https://github.com/D12126977/survey.
[5] Surafel Lemma Abebe, Sonia Haiduc, Paolo Tonella, and Andrian Marcus. 2009. Lexicon bad smells in software. In *2009 16th Working Conference on Reverse Engineering (WCRE'09)*. IEEE Computer Society, Washington, DC, 95–99. DOI:https://doi.org/10.1109/WCRE.2009.26
[6] Surafel Lemma Abebe and Paolo Tonella. 2010. Natural language parsing of program element names for concept extraction. In *IEEE International Conference on Program Comprehension*. 156–159.
[7] S. L. Abebe and P. Tonella. 2013. Automated identifier completion and replacement. In *17th European Conference on Software Maintenance and Reengineering (CSMR'13)*. 263–272. DOI:https://doi.org/10.1109/CSMR.2013.35
[8] Alex Aiken. 2005. MOSS, a system for detecting software plagiarism. University of California, Berkeley. 9.
[9] Nouh Alhindawi, Natalia Dragan, Michael L. Collard, and Jonathan I. Maletic. 2013. Improving feature location by enhancing source code with stereotypes. In *IEEE International Conference on Software Maintenance*. 300–309.
[10] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2014. Learning natural coding conventions. In *22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, 281–293. DOI:https://doi.org/10.1145/2635868.2635883
[11] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2015. Suggesting accurate method and class names. In *2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, 38–49. DOI:https://doi.org/10.1145/2786805.2786849
[12] Reem Saleh Alsuhaibani. 2015. Part-of-speech tagging of source code identifiers using programming language context versus natural language context. *Dissertations & Theses - Gradworks* (2015).
[13] Reem S. Alsuhaibani, Christian D. Newman, Michael L. Collard, and Jonathan I. Maletic. 2015. Heuristic-based part-of-speech tagging of source code identifiers and comments. In *2015 IEEE 5th Workshop on Mining Unstructured Data (MUD)*. IEEE, 1–6.
[14] Everton L. G. Alves, Patrícia D. L. Machado, Tiago Massoni, and Miryung Kim. 2016. Prioritizing test cases for early detection of refactoring faults. *Software Testing Verification & Reliability* 26, 5 (2016).
[15] N. Anand. 1988. Clarify function. *ACM SIGPLAN Notices* 23, 6 (1988), 69–79.
[16] Nicolas Anquetil and Timothy Lethbridge. 1998. Assessing the relevance of identifier names in a legacy software system. In *Proceedings of the 1998 Conference of the Centre for Advanced Studies on Collaborative Research*. IBM Press, 4.
[17] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. 2000. Recovering traceability links between code and documentation. In *International Conference on Software Maintenance*. 40.
[18] Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. 2016. Linguistic antipatterns: What they are and how developers perceive them. *Empirical Softw. Eng.* 21, 1 (Feb. 2016), 104–158. DOI:https://doi.org/10.1007/s10664-014-9350-8
[19] Venera Arnaoudova, Massimiliano Di Penta, Giuliano Antoniol, and Yann-Gael Gueheneuc. 2013. A new family of software anti-patterns: Linguistic anti-patterns. In *2013 17th European Conference on Software Maintenance and Reengineering (CSMR'13)*. IEEE Computer Society, Washington, DC, 187–196. DOI:https://doi.org/10.1109/CSMR.2013.28

[20] V. Arnaoudova, L. M. Eshkevari, M. Di Penta, R. Oliveto, G. Antoniol, and Y. G. Guéhéneuc. 2014. REPENT: Analyzing the nature of identifier renamings. *IEEE Transactions on Software Engineering* 40, 5 (May 2014), 502–532. DOI: https://doi.org/10.1109/TSE.2014.2312942

[21] Ricardo A. Baeza-Yates and Chris H. Perleberg. 1996. Fast and practical approximate string matching. *Inf. Process. Lett.* 59, 1 (July 1996), 21–27. DOI: https://doi.org/10.1016/0020-0190(96)00083-X

[22] Dave Binkley, Matthew Hearn, and Dawn Lawrie. 2011. Improving identifier informativeness using part of speech information. In *Working Conference on Mining Software Repositories*. 203–206.

[23] Dave Binkley and Dawn Lawrie. 2015. The impact of vocabulary normalization. *Journal of Software Evolution & Process* 27, 4 (2015), 255–273.

[24] Thorsten Brants. 2000. TnT: A statistical part-of-speech tagger. In *Conference on Applied Natural Language Processing*. 224–231.

[25] A. Broder. 1997. On the resemblance and containment of documents. In *Compression and Complexity of Sequences 1997 (SEQUENCES'97)*. IEEE Computer Society, Washington, DC, 21–29. http://dl.acm.org/citation.cfm?id=829502.830043

[26] Christopher J. C Burges. 1998. A tutorial on support vector machines for pattern recognition. *Data Mining & Knowledge Discovery* 2, 2 (1998), 121–167.

[27] Burgula, Venkatesh Reddy. 2015. *An Automated Rename Refactoring for Go*. Master's Thesis. Auburn University.

[28] Simon Butler. 2009. The effect of identifier naming on source code readability and quality. In *Doctoral Symposium for ESEC/FSE on Doctoral Symposium (ESEC/FSE Doctoral Symposium'09)*. ACM, New York, 33–34. DOI: https://doi.org/10.1145/1595782.1595796

[29] Simon Butler. 2011. *INTT: Identifier name tokenisation tool. Open University* (2011).

[30] Simon Butler. 2012. Mining Java class identifier naming conventions. In *34th International Conference on Software Engineering (ICSE'12)*. IEEE Press, Piscataway, NJ, 1641–1643. http://dl.acm.org/citation.cfm?id=2337223.2337509.

[31] Simon Butler. 2016. *Analysing Java Identifier Names*. Ph.D. Dissertation. Department of Computing and Communications Faculty of Mathematics, Computing and Technology, The Open University.

[32] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. 2009. Relating identifier naming flaws and code quality: An empirical study. In *2009 16th Working Conference on Reverse Engineering*. IEEE, 31–35.

[33] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. 2011. Mining Java class naming conventions. In *2011 27th IEEE International Conference on Software Maintenance (ICSM'11)*. IEEE Computer Society, Washington, DC, 93–102. DOI: https://doi.org/10.1109/ICSM.2011.6080776

[34] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. 2013. INVocD: Identifier name vocabulary dataset. In *Working Conference on Mining Software Repositories*. 405–408.

[35] L. W. Cannon, R. A. Elliott, L. W. Kirchhoff, J. H. Miller, J. M. Milner, R. W. Mitze, E. P. Schan, N. O. Whittington, Henry Spencer, David Keppel, Mark Brader, and Diomidis Spinellis. 1991. *Recommended C Style and Coding Standards*. Pocket reference guide. Specialized Systems Consultants.

[36] B. Caprile and P. Tonella. 2000. Restructuring program identifier names. In *International Conference on Software Maintenance, 2000*. 97–107. DOI: https://doi.org/10.1109/ICSM.2000.883022

[37] B. Caprile and P. Tonella. 1999. Nomen est omen: Analyzing the language of function identifiers. In *6th Working Conference on Reverse Engineering, 1999*. 112–122. DOI: https://doi.org/10.1109/WCRE.1999.806952

[38] Breck Carter. 1982. On choosing identifiers. *SIGPLAN Notes* 17, 5 (May 1982), 54–59. DOI: https://doi.org/10.1145/947923.947930

[39] Nuno Ramos Carvalho, Jose Joao Almeida, Pedro Rangel Henriques, and Maria Joao Varanda. 2015. From source code identifiers to natural language terms. *Journal of Systems & Software* 100 (2015), 117–128.

[40] S. Ceri, G. Gottlob, and L. Tanca. 2002. What you always wanted to know about datalog (and never dared to ask). *IEEE Transactions on Knowledge & Data Engineering* 1, 1 (2002), 146–166.

[41] Nicholas Chen and Ralph Johnson. 2008. *Toward Refactoring in a Polyglot World: Extending Automated Refactoring Support Across Java and XML*. 4:1–4:4 pages.

[42] Kenneth Ward Church and Jonathan Isaac Helfman. 1993. Dotplot: A program for exploring self-similarity in millions of lines of text and code. *Journal of Computational & Graphical Statistics* 2, 2 (1993), 153–174.

[43] Julien Cohen. 2016. Renaming global variables in C mechanically proved correct. 216, *Proc. VPT 2016* (2016), 50–64.

[44] Michael L. Collard, Michael John Decker, and Jonathan I. Maletic. 2013. srcML: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration. In *IEEE International Conference on Software Maintenance*. 516–519.

[45] F. Corbo, C. del Grosso, and M. di Penta. 2007. Smart formatter: Learning coding style from existing source code. In *IEEE International Conference on Software Maintenance (ICSM'07)*. 525–526. DOI: https://doi.org/10.1109/ICSM.2007.4362682

[46] Maartje De Jonge and Eelco Visser. 2012. A language generic solution for name binding preservation in refactorings. In *12th Workshop on Language Descriptions, Tools, and Applications*. 1–8.

[47] Andrea De Lucia, Massimiliano Di Penta, and Rocco Oliveto. 2011. Improving source code lexicon via traceability and information retrieval. *IEEE Trans. Softw. Eng.* 37, 2 (Mar. 2011), 205–227. DOI: https://doi.org/10.1109/TSE.2010.89

[48] Scott Deerwester, Susan T. Dumais, George W. Furnas, Thomas K. Landauer, and Richard Harshman. 1990. Indexing by latent semantic analysis. *Journal of the Association for Information Science and Technology* 41, 6 (1990), 391–407.

[49] Florian Deissenboeck and Markus Pizka. 2005. Concise and consistent naming. *Software Quality Journal* 14, 3 (2005), 261–282.

[50] Florian Deissenboeck and Daniel Ratiu. 2006. A unified meta-model for concept-based reverse engineering. In *3rd International Workshop on Metamodels, Schemas, Grammars and Ontologies (ATEM'06).*

[51] Sergio Di Martino, Valerio Maggio, and Anna Corazza. 2012. LINSEN: An efficient approach to split identifiers and expand abbreviations. In *2012 IEEE International Conference on Software Maintenance (ICSM'12).* IEEE Computer Society, Washington, DC, 233–242. DOI: https://doi.org/10.1109/ICSM.2012.6405277

[52] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. 2005. Automatic detection of refactorings for libraries and frameworks. In *Proceedings of Workshop on Object Oriented Reengineering (WOOR'05).*

[53] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. 2006. Automated detection of refactorings in evolving components. In *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP'06).* Springer-Verlag, Berlin, Heidelberg, 404–428. DOI: https://doi.org/10.1007/11785477_24

[54] M Dorigo, V Maniezzo, and A Colorni. 1996. Ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems Man & Cybernetics Part B Cybernetics A Publication of the IEEE Systems Man & Cybernetics Society* 26, 1 (1996), 29.

[55] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker. 2009. Mining source code to automatically split identifiers for software analysis. In *6th IEEE International Working Conference on Mining Software Repositories (MSR'09).* 71–80. DOI: https://doi.org/10.1109/MSR.2009.5069482

[56] Laleh Mousavi Eshkevari. 2015. *Automatic Detection and Classification of Identifier Renamings.* Ph.D. Dissertation. The School of the Thesis.

[57] Laleh M. Eshkevari, Venera Arnaoudova, Massimiliano Di Penta, Rocco Oliveto, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. 2011. An exploratory study of identifier renamings. In *8th Working Conference on Mining Software Repositories.* ACM, 33–42.

[58] J. R. Falleri, M. Huchard, M. Lafourcade, C. Nebut, V. Prince, and M. Dao. 2010. Automatic extraction of a WordNet-like identifier network from software. In *International Conference on Program Comprehension.* 4–13.

[59] Henry Feild, David Binkley, and Dawn Lawrie. 2006. An empirical comparison of techniques for extracting concept abbreviations from identifiers. In *IASTED International Conference on Software Engineering and Applications (SEA'06).*

[60] Asger Feldthaus, Todd Millstein, Anders Møller, Max Schäfer, and Frank Tip. 2011. Tool-supported refactoring for JavaScript. In *2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'11).* ACM, New York, 119–138. DOI: https://doi.org/10.1145/2048066.2048078

[61] Asger Feldthaus and Anders Møller. 2013. Semi-automatic rename refactoring for JavaScript. *SIGPLAN Notes* 48, 10 (Oct. 2013), 323–338. DOI: https://doi.org/10.1145/2544173.2509520

[62] Christiane Fellbaum. 2012. *WordNet.* Blackwell Publishing Ltd. 231–243.

[63] Beat Fluri, Michael Wuersch, Martin PInzger, and Harald Gall. 2007. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering* 33, 11 (2007), 725–743.

[64] Ernest J. Friedman-Hill. 1997. *JESS, the Java expert system shell. Office of Scientific & Technical Information Technical Reports* (1997).

[65] Jianfeng Gao, Ming Zhou, Jian Yun Nie, Hongzhao He, and Weijun Chen. 2002. Resolving query translation ambiguity using a decaying co-occurrence model and syntactic dependence relations. In *Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'02).* Association for Computing Machinery, New York, NY, USA, 183–190. DOI: https://doi.org/10.1145/564376.564409

[66] Xi Ge, Quinton L. DuBose, and Emerson Murphy-Hill. 2012. Reconciling manual and automatic refactoring. In *34th International Conference on Software Engineering (ICSE'12).* IEEE Press, Piscataway, NJ, 211–221. http://dl.acm.org/citation.cfm?id=2337223.2337249

[67] Joseph (Yossi) Gil and Itay Maman. 2005. Micro patterns in Java code. In *20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05).* ACM, New York, 97–116. DOI: https://doi.org/10.1145/1094811.1094819

[68] M. W. Godfrey and L. Zou. 2005. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering* 31, 2 (Feb. 2005), 166–181. DOI: https://doi.org/10.1109/TSE.2005.28

[69] Thomas R. G. Green. 1989. Cognitive dimensions of notations. *People and Computers V* (1989), 443–460.

[70] L. Guerrouj. 2013. Normalizing source code vocabulary to support program comprehension and software quality. (2013), 1385–1388.

[71] Latifa Guerrouj, Massimiliano Di Penta, Giuliano Antoniol, and Yann-Gaél Guéhéneuc. 2013. TIDIER: An identifier splitting approach using speech recognition techniques. *Journal of Software: Evolution and Process* 25, 6 (2013), 575–599. DOI: https://doi.org/10.1002/smr.539

[72] Latifa Guerrouj, Philippe Galinier, Yann Gael Gueheneuc, Giuliano Antoniol, and Massimiliano Di Penta. 2012. TRIS: A fast and accurate identifiers splitting and expansion algorithm. In *Proceedings of the 19th Working Conference on Reverse Engineering*. 103–112.

[73] Xinping Guo, James R. Cordy, and Thomas R. Dean. 2003. Unique renaming of Java using source transformation. In *IEEE International Workshop on Source Code Analysis & Manipulation*.

[74] S. Gupta, S. Malik, L. Pollock, and K. Vijay-Shanker. 2013. Part-of-speech tagging of program identifiers for improved text-based software engineering tools. In *IEEE 21st International Conference on Program Comprehension (ICPC'13)*. 3–12. DOI: https://doi.org/10.1109/ICPC.2013.6613828

[75] Yoshiki Higo and Shinji Kusumoto. 2014. MPAnalyzer: A tool for finding unintended inconsistencies in program source code. In *ACM/IEEE International Conference on Automated Software Engineering*. 843–846.

[76] Emily Hill, David Binkley, Dawn Lawrie, Lori Pollock, and K. Vijay-Shanker. 2014. An empirical study of identifier splitting techniques. *Empirical Software Engineering* 19, 6 (2014), 1754–1780.

[77] Emily Hill, Zachary P. Fry, Haley Boyd, Giriprasad Sridhara, Yana Novikova, Lori Pollock, and K. Vijay-Shanker. 2008. AMAP: Automatically mining abbreviation expansions in programs to enhance software maintenance tools. In *2008 International Working Conference on Mining Software Repositories (MSR'08)*. ACM, New York, 79–88. DOI: https://doi.org/10.1145/1370750.1370771

[78] Einar W. Høst and Bjarte M. Østvold. 2007. The programmer's lexicon, Volume I: The verbs. In *7th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'07)*. 193–202. DOI: https://doi.org/10.1109/SCAM.2007.18

[79] Einar W. Høst and Bjarte M. Østvold. 2008. The Java programmer's phrase book. In *International Conference on Software Language Engineering*. Springer, 322–341.

[80] Einar W. Høst and Bjarte M. Østvold. 2009. In *23rd European Conference on Object-Oriented Programming (ECOOP'09)*. Springer Berlin, Berlin, Chapter Debugging Method Names, 294–317. DOI: https://doi.org/10.1007/978-3-642-03013-0_14

[81] Michael Hucka. 2018. Spiral: Splitters for identifiers in source code files. *Journal of Open Source Software* 3, 24 (2018), 653.

[82] T. Imielienskin, A. Swami, and R. Agrawal. 1993. Mining association rules between set of items in large databases. *ACM SIGMOD Record* 22, 2 (1993).

[83] Patricia Jablonski and Daqing Hou. 2007. CReN: A tool for tracking copy-and-paste code clones and renaming identifiers consistently in the IDE. In *2007 OOPSLA Workshop on Eclipse Technology eXchange (ECLIPSE'07)*. ACM, New York, 16–20. DOI: https://doi.org/10.1145/1328279.1328283

[84] Patricia Jablonski and Daqing Hou. 2010. Renaming parts of identifiers consistently within code clones. In *2010 IEEE 18th International Conference on Program Comprehension (ICPC'10)*. IEEE Computer Society, Washington, DC, 38–39. DOI: https://doi.org/10.1109/ICPC.2010.23

[85] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. DECKARD: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE Computer Society, Washington, DC, 96–105. DOI: https://doi.org/10.1109/ICSE.2007.30

[86] Lingxiao Jiang, Zhendong Su, and Edwin Chiu. 2007. Context-based detection of clone-related bugs. In *6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE'07)*. ACM, New York, 55–64. DOI: https://doi.org/10.1145/1287624.1287634

[87] Y. Jiang, H. Liu, J. Q. Zhu, and L. Zhang. 2018. Automatic and accurate expansion of abbreviations in parameters. In *IEEE Transactions on Software Engineering*. DOI: 10.1109/TSE.2018.2868762

[88] Derek M. Jones. 2003. *The New C Standard: A Cultural and Economic Commentary*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA.

[89] Toshihiro Kamiya. 2008. Variation analysis of context-sharing identifiers with code clones. In *IEEE International Conference on Software Maintenance*. 464–465.

[90] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* 28, 7 (2002), 654–670.

[91] Edvard K. Karlsen, Einar W. Høst, and Bjarte M. Østvold. 2012. Finding and fixing Java naming bugs with the Lancelot eclipse plugin. In *ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation (PEPM'12)*. ACM, New York, 35–38. DOI: https://doi.org/10.1145/2103746.2103756

[92] Yuki Kashiwabara, Takashi Ishio, Hideaki Hata, and Katsuro Inoue. 2015. Method verb recommendation using association rule mining in a set of existing projects. *IEICE Trans. Inf. Syst.* 98, 3 (2015), 627–636.

[93] Yuki Kashiwabara, Yuya Onizuka, Takashi Ishio, Yasuhiro Hayase, Tetsuo Yamamoto, and Katsuro Inoue. 2014. Recommending verbs for rename method using association rule mining. In *Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering*. 323–327.

[94] David Kawrykow and Martin P. Robillard. 2011. Non-essential changes in version histories. In *33rd International Conference on Software Engineering (ICSE'11)*. ACM, New York, 351–360. DOI: https://doi.org/10.1145/1985793.1985842

[95] Martin Kempf, Reto Kleeb, Michael Klenk, and Peter Sommerlad. 2008. Cross language refactoring for Eclipse plug-ins. In *Workshop on Refactoring Tools*. 1–4.

[96] Suntae Kim and Dongsun Kim. 2016. Automatic identifier inconsistency detection using code dictionary. *Empirical Software Engineering* 21, 2 (2016), 565–604.

[97] Sunghun Kim, Kai Pan, and E. James Whitehead, Jr.2005. When functions change their names: Automatic detection of origin relationships. In *12th Working Conference on Reverse Engineering (WCRE'05)*. IEEE Computer Society, Washington, DC, 143–152. DOI: https://doi.org/10.1109/WCRE.2005.33

[98] T. Y. Kim, S. Kim, J. A. Kim, J. Y. Choi, J. H. Lee, Y. Cho, and Y. K. Nam. 2018. Automatic identification of Java method naming patterns using cascade K-medoids. *KSII Transactions on Internet & Information Systems* 12, 2 (2018), 873–891.

[99] Adrian Kuhn. 2010. On recommending meaningful names in source and UML. In *2nd International Workshop on Recommendation Systems for Software Engineering*. 50–51.

[100] Shinji Kusumoto and Yoshiki Higo. 2012. How often do unintended inconsistencies happen? Deriving modification patterns and detecting overlooked code fragments. In *2012 IEEE International Conference on Software Maintenance (ICSM'12)*. IEEE Computer Society, Washington, DC, 222–231. DOI: https://doi.org/10.1109/ICSM.2012.6405275

[101] John D. Lafferty, Andrew McCallum, and Fernando C. N. Pereira. 2001. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *18th International Conference on Machine Learning (ICML'01)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 282–289. http://dl.acm.org/citation.cfm?id=645530.655813

[102] Dawn Lawrie and Dave Binkley. 2011. Expanding identifiers to normalize source code vocabulary. In *2011 27th IEEE International Conference on Software Maintenance (ICSM'11)*. IEEE Computer Society, Washington, DC, 113–122. DOI: https://doi.org/10.1109/ICSM.2011.6080778

[103] Dawn Lawrie, Dave Binkley, and Christopher Morrell. 2010. Normalizing source code vocabulary. In *2010 17th Working Conference on Reverse Engineering (WCRE'10)*. IEEE Computer Society, Washington, DC, 3–12. DOI: https://doi.org/10.1109/WCRE.2010.10

[104] D. Lawrie, H. Feild, and D. Binkley. 2006. Syntactic identifier conciseness and consistency. In *6th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'06)*. 139–148. DOI: https://doi.org/10.1109/SCAM.2006.31

[105] D. Lawrie, H. Feild, and D. Binkley. 2007. Extracting meaning from abbreviated identifiers. In *7th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'07)*. 213–222. DOI: https://doi.org/10.1109/SCAM.2007.17

[106] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. 2007. Effective identifier names for comprehension and memory. *Innovations in Systems & Software Engineering* 3, 4 (2007), 303–318.

[107] Quoc Le and Tomas Mikolov. 2014. Distributed representations of sentences and documents. In *International Conference on Machine Learning*. 1188–1196.

[108] V. I. Levenshtein. 1966. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady* 10, 1 (1966), 707–710.

[109] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. 2006. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Trans. Softw. Eng.* 32, 3 (March 2006), 176–192. DOI: https://doi.org/10.1109/TSE.2006.28

[110] Ben Liblit, Andrew Begel, and Sweetser. 2006. Cognitive perspectives on the role of naming in computer programs. In *Annual Psychology of Programming Workshop* (2006).

[111] Bin Lin, Simone Scalabrino, Andrea Mocci, Rocco Oliveto, Gabriele Bavota, and Michele Lanza. 2017. Investigating the use of code analysis and NLP to promote a consistent usage of identifiers. In *IEEE International Working Conference on Source Code Analysis and Manipulation*. 81–90.

[112] Dekan Lin. 1999. MINIPAR: A minimalist parser. *Maryland Linguistics Colloquium* (1999).

[113] Panagiotis K. Linos. 1995. Polycare: A tool for re-engineering multi-language program integrations. In *1st IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'95)*. IEEE, 338–341.

[114] H. Liu, Q. Liu, Y. Liu, and Z. Wang. 2015. Identifying renaming opportunities by expanding conducted rename refactorings. *IEEE Transactions on Software Engineering* 41, 9 (Sept 2015), 887–900. DOI: https://doi.org/10.1109/TSE.2015.2427831

[115] Hui Liu, Qiurong Liu, Cristian Alexandru Staicu, Michael Pradel, and Yue Luo. 2016. Nomen est omen: Exploring and exploiting similarities between argument and parameter names. In *ACM 38th IEEE International Conference on Software Engineering*.

[116] Kui Liu and Dongsun Kim. 2019. Learning to spot and refactor inconsistent method names. In *International Conference on Software Engineering*.

[117] Edward Loper and Steven Bird. 2002. NLTK: The natural language toolkit. In *ACL Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics. Association for Computational Linguistics, Philadelphia, Pa.*

[118] Nioosha Madani. 2010. *Heuristic Splitting of Source Code Identifiers.* Master's Thesis. Ecole Polytechnique de Montreal, Canada.

[119] N. Madani, L. Guerrouj, M. Di Penta, Y. G. Gueheneuc, and G. Antoniol. 2010. Recognizing words from source code identifiers using speech recognition techniques. In *14th European Conference on Software Maintenance and Reengineering (CSMR'10).* 68–77. DOI: https://doi.org/10.1109/CSMR.2010.31

[120] G. Malpohl, J. J. Hunt, and W. F. Tichy. 2000. Renaming detection. In *15th IEEE International Conference on Automated Software Engineering (ASE'00).* 73–80. DOI: https://doi.org/10.1109/ASE.2000.873652

[121] Masakazu Matsugu, Katsuhiko Mori, Yusuke Mitari, and Yuji Kaneda. 2003. Subject independent facial expression recognition with robust face detection using a convolutional neural network. *Neural Networks* 16, 5–6 (2003), 555–559.

[122] Philip Mayer, Michael Kirsch, and Minh Anh Le. 2017. On multi-language software development, cross-language links and accompanying tools: A survey of professional software developers. *Journal of Software Engineering Research & Development* 5, 1 (2017), 1.

[123] Philip Mayer and Andreas Schroeder. 2013. Towards automated cross-language refactorings between Java and DSLs used by Java frameworks. In *2013 ACM Workshop on Workshop on Refactoring Tools.* ACM, 5–8.

[124] Philip Mayer and Andreas Schroeder. 2014. Automated multi-language artifact binding and rename refactoring between Java and DSLs used by Java frameworks. In *European Conference on Object-Oriented Programming (ECOOP'14)*, Richard Jones (Ed.). Springer Berlin, Berlin, Germany, 437–462.

[125] Steve McConnell. 2004. *Code Complete.* Pearson Education.

[126] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[127] George A. Miller. 1995. WordNet: A lexical database for English. *Communications of the ACM* 38 (1995), 39–41.

[128] G. C. Murphy, M. Kersten, and L. Findlater. 2006. How are Java software developers using the Elipse IDE? *IEEE Software* 23, 4 (July 2006), 76–83. DOI: https://doi.org/10.1109/MS.2006.105

[129] E. Murphy-Hill. 2008. Breaking the barriers to successful refactoring: Observations and tools for extract method. In *Proceedings of the 30th International Conference on Software Engineering ACM New York, NY, USA, 2008.*

[130] Emerson Murphy-Hill and Andrew P. Black. 2008. Refactoring tools: Fitness for purpose. *IEEE Software* 25, 5 (2008), 38–44.

[131] Iulian Neamtiu, Jeffrey S. Foster, and Michael Hicks. 2005. Understanding source code evolution using abstract syntax tree matching. *ACM SIGSOFT Software Engineering Notes* 30, 4 (2005), 1–5.

[132] Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E. Johnson, and Danny Dig. 2013. A comparative study of manual and automated refactorings. In *European Conference on Object-Oriented Programming*. 552–576.

[133] Christian D. Newman, Reem S. Alsuhaibani, Michael L. Collard, and Jonathan I. Maletic. 2017. Lexical categories for source code identifiers. In *IEEE International Conference on Software Analysis, Evolution and Reengineering*. 228–239.

[134] H. Ney. 1984. The use of a one-stage dynamic programming algorithm for connected word recognition. *IEEE Transactions on Acoustics Speech & Signal Processing* 32, 2 (1984), 263–271.

[135] Hung Viet Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. 2011. Auto-locating and fix-propagating for HTML validation errors to PHP server-side code. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE'11).* IEEE Computer Society, Washington, DC, 13–22. DOI: https://doi.org/10.1109/ASE.2011.6100047

[136] Hung Viet Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and T. N. Nguyen. 2012. BabelRef: Detection and renaming tool for cross-language program entities in dynamic web applications. In *International Conference on Software Engineering*. 1391–1394.

[137] S. Nissen. 2005. Neural Networks Made Simple. Software 2.0. 2, 14–19.

[138] Wyatt Olney, Emily Hill, Chris Thurber, and Bezalem Lemma. 2016. Part of speech tagging Java method names. In *IEEE International Conference on Software Maintenance and Evolution*. 483–487.

[139] F. William Opdyke. 1992. *Refactoring Object-Oriented Frameworks. Ph.D. Thesis, University of Illinois at Urbana-Champaign* (1992).

[140] Jeffrey L. Overbey and Ralph E. Johnson. 2011. Differential precondition checking: A lightweight, reusable analysis for refactoring tools. In *IEEE/ACM International Conference on Automated Software Engineering*. 303–312.

[141] Jeffrey L. Overbey, Ralph E. Johnson, and Munawar Hafiz. 2016. Differential precondition checking: A language-independent, reusable analysis for refactoring engines. *Automated Software Engineering* 23, 1 (2016), 77–104.

[142] University of Pennsylvania. 2015. *Linguistic Data Consortium.* (2015).
[143] Anthony Peruma, Mohamed Wiem Mkaouer, Michael J. Decker, and Christian D. Newman. 2018. An empirical investigation of how and why developers rename identifiers. In *Proceedings of the 2nd International Workshop on Refactoring (IWoR'18).* ACM, New York, 26–33. DOI: https://doi.org/10.1145/3242163.3242169
[144] P. Pirapuraj and Indika Perera. 2017. Analyzing source code identifiers for code reuse using NLP techniques and WordNet. In *Moratuwa Engineering Research Conference.*
[145] D. Ratiu and F. Deissenboeck. 2006. How programs represent reality (and how they don't). In *13th Working Conference on Reverse Engineering (WCRE'06).* 83–92. DOI: https://doi.org/10.1109/WCRE.2006.32
[146] B. Ray, M. Kim, S. Person, and N. Rungta. 2013. Detecting and characterizing semantic inconsistencies in ported code. In *IEEE/ACM 28th International Conference on Automated Software Engineering (ASE'13).* 367–377. DOI: https://doi.org/10.1109/ASE.2013.6693095
[147] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting program properties from "Big Code". *SIGPLAN Not.* 50, 1 (Jan. 2015), 111–124. DOI: https://doi.org/10.1145/2775051.2677009
[148] Steven P. Reiss. 2007. Automatic code stylizing. In *IEEE/ACM International Conference on Automated Software Engineering.* 74–83.
[149] Marjorie Richardson. 1999. ispell: Spelling checker. (1999).
[150] Juergen Rilling and Tuomas Klemola. 2003. Identifying comprehension bottlenecks using program slicing and cognitive complexity metrics. In *11th IEEE International Workshop on Program Comprehension, 2003.* IEEE, 115–124.
[151] Donald Bradley Roberts. 1999. *Practical Analysis for Refactoring.* Ph.D. Dissertation. University of Illinois at Urbana-Champaign, USA. Advisor(s) Ralph Johnson. Order Number: AAI9944985.
[152] Max Schäfer, Torbjörn Ekman, and Oege de Moor. 2008. Sound and extensible renaming for Java. In *23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'08).* ACM, New York, 277–294. DOI: https://doi.org/10.1145/1449764.1449787
[153] Andrea Schankin, Annika Berger, Daniel V. Holt, Johannes C. Hofmeister, Till Riedel, and Michael Beigl. 2018. Descriptive compound identifier names improve source code comprehension. In *26th Conference on Program Comprehension (ICPC'18).* ACM, New York, 31–40. DOI: https://doi.org/10.1145/3196321.3196332
[154] Hagen Schink and Martin Kuhlemann. 2010. *Hurdles in Refactoring Multi-Language Programs. Technical Report FIN-007* (2010).
[155] Hagen Schink, Martin Kuhlemann, Gunter Saake, and Ralf Lammel. 2011. Hurdles in multi-language refactoring of hibernate applications. In *Proceedings of the International Conference on Software and Data Technologies (ICSOFT'11).* 129–134.
[156] TreeTagger. 2009. https://www.cis.uni-muenchen.de/~schmid/tools/TreeTagger/.
[157] SCOWL. 2017. http://wordlist.aspell.net/scowl-readme/.
[158] David Shepherd, Lori Pollock, and K. Vijay-Shanker. 2007. Case study: Supplementing program analysis with natural language analysis to improve a reverse engineering task. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering.* 49–54.
[159] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why we refactor? Confessions of GitHub contributors. In *2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'16).* ACM, New York, 858–870. DOI: https://doi.org/10.1145/2950290.2950305
[160] J. Singer and C. Kirkham. 2008. Exploiting the correspondence between micro patterns and class names. In *2008 8th IEEE International Working Conference on Source Code Analysis and Manipulation.* 67–76. DOI: https://doi.org/10.1109/SCAM.2008.23
[161] H. M. Sneed. 1996. Object-oriented COBOL recycling. In *3rd Working Conference on Reverse Engineering, 1996.* 169–178.
[162] Gustavo Soares, Bruno Catao, Catuxe Varjao, Solon Aguiar, Rohit Gheyi, and Tiago Massoni. 2011. Analyzing refactorings on software repositories. In *25th Brazilian Symposium on Software Engineering, SBES 2011, Sao Paulo, Brazil, September 28-30, 2011.* 164–173. DOI: https://doi.org/10.1109/SBES.2011.21
[163] Gustavo Soares, Rohit Gheyi, Dalton Serey, and Tiago Massoni. 2010. Making program refactoring safer. *IEEE Software* 27, 4 (2010), 52–57.
[164] Luis Enrique Sucar. 2015. *Probabilistic Graphical Models.* Springer London. 149–151 pages.
[165] Ashish Sureka. 2012. Source code identifier splitting using Yahoo image and web search engine. In *International Workshop on Software Mining.* 1–8.
[166] Takayuki Suzuki, Kazunori Sakamoto, Fuyuki Ishikawa, and Shinichi Honiden. 2014. An approach for evaluating and suggesting method names using N-gram models. In *22nd International Conference on Program Comprehension (ICPC'14).* ACM, New York, 271–274. DOI: https://doi.org/10.1145/2597008.2597797
[167] Armstrong A. Takang, Penny A. Grubb, and Robert D. Macredie. 1996. The effects of comments and identifier names on program comprehensibility: An experimental investigation. *J. Prog. Lang.* 4, 3 (1996), 143–167.

[168] Kunal Taneja, Danny Dig, and Tao Xie. 2007. Automated detection of API refactorings in libraries. In *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*. ACM, New York, 377–380. DOI: https://doi.org/10.1145/1321631.1321688

[169] Andreas Thies and Christian Roth. 2010. Recommending rename refactorings. In *2nd International Workshop on Recommendation Systems for Software Engineering (RSSE'10)*. ACM, New York, 1–5. DOI: https://doi.org/10.1145/1808920.1808921

[170] Kristina Toutanova, Dan Klein, Christopher D. Manning, and Yoram Singer. 2003. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology*. 173–180.

[171] Watkins Trevor. 1996. ML for the working programmer. (1996).

[172] Qiang Tu and Michael W. Godfrey. 2002. An integrated approach for studying architectural evolution. In *International Workshop on Program Comprehension, 2002*. 127–136.

[173] F. Van Rysselberghe and S. Demeyer. 2003. Reconstruction of successful software evolution using clone detection. In *Proceedings of the Sixth International Workshop on Principles of Software Evolution*. 126–130.

[174] Mathieu Verbaere, Ettinger Ran, and Oege De Moor. 2006. JunGL: A scripting language for refactoring. In *International Conference on Software Engineering*. 172–181.

[175] Amit Verma, Srishti Gupta, and Iqbaldeep Kaur. 2016. Inconsistency detection in software component source code using ant colony optimization and neural network algorithm. *Indian Journal of Science & Technology* 9, 40 (2016).

[176] Peter Weissgerber and Stephan Diehl. 2006. Identifying refactorings from source-code changes. In *IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. IEEE Computer Society, Washington, DC, 231–240. DOI: https://doi.org/10.1109/ASE.2006.41

[177] Zhenchang Xing and Eleni Stroulia. 2005. UMLDiff: An algorithm for object-oriented design differencing. In *IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE Computer Scociety, Washington, DC* (2005), 54–65.

[178] Zhenchang Xing and Eleni Stroulia. 2006. Refactoring detection based on UMLDiff change-facts queries. In *13th Working Conference on Reverse Engineering (WCRE'06)*. IEEE Computer Society, Washington, DC, USA, 263–274. DOI: https://doi.org/10.1109/WCRE.2006.48

[179] X. Yan. 2003. CloSpan: Mining closed sequential patterns in large datasets. In *2003 SIAM International Conference on Data Mining* (2003), 166–177.

[180] Deheng Ye, Zhenchang Xing, Jing Li, and Nachiket Kapre. 2016. Software-specific part-of-speech tagging: An experimental study on stack overflow. In *31st Annual ACM Symposium on Applied Computing (SAC'16)*. ACM, New York, 1378–1385. DOI: https://doi.org/10.1145/2851613.2851772

[181] Shusi Yu, Ruichang Zhang, and Jihong Guan. 2012. Properly and automatically naming Java methods: A machine learning based approach. In *International Conference on Advanced Data Mining and Applications*. Springer, 235–246.

Received February 2019; revised September 2019; accepted January 2020