Automatic and Accurate Expansion of Abbreviations in Parameters

Yanjie Jiang, Hui Liu, Jiaqi Zhu, and Lu Zhang

Abstract—Abbreviations are widely used in identifiers. However, they have severe negative impact on program comprehension and IR-based software maintenance activities, e.g., concept location, software clustering, and recovery of traceability links. Consequently, a number of efficient approaches have been proposed successfully to expand abbreviations in identifiers. Most of such approaches rely heavily on dictionaries, and rarely exploit the specific and fine-grained context of identifiers. As a result, such approaches are less accurate in expanding abbreviations (especially short ones) that may match multiple dictionary words. To this end, in this paper we propose an automatic approach to improve the accuracy of abbreviation expansion by exploiting the specific and fine-grained context. It focuses on a special but common category of abbreviations (abbreviations in parameter names), and thus it can exploit the specific and fine-grained context, i.e., the type of the enclosing parameter as well the corresponding formal (or actual) parameter name. The recent empirical study on parameters suggest that actual parameters are often lexically similar to their corresponding formal parameters. Consequently, it is likely that an abbreviation in a formal parameter can find its full terms in the corresponding actual parameter, and vice versa. Based on this assumption, a series of heuristics are proposed to look for full terms from the corresponding actual (or formal) parameter names. To the best of our knowledge, we are the first to expand abbreviations by exploiting the lexical similarity between actual and formal parameters. We also search for full terms in the data type of the enclosing parameter. Only if all such heuristics fail, the approach turns to the traditional abbreviation dictionaries. We evaluate the proposed approach on seven well known open-source projects. Evaluation results suggest that when only parameter abbreviations are involved, the proposed approach can improve the precision from 26 to 95 percent and recall from 26 to 65 percent compared against the state-of-the-art general purpose approach. Consequently, the proposed approach could be employed as a useful supplement to existing approaches to expand parameter abbreviations.

Index Terms—Abbreviation, expansion, comprehension, lexical similarity, quality, information retrieval

1 Introduction

High quality names of software entities are crucial for the readability and maintainability of programs [1]. Such names, usually called identifiers, account for 70 percent of source code in terms of characters [2]. Consequently, the readability of such identifiers are crucial for the readability of software programs, especially complex programs [3].

Software identifiers often contain abbreviations. However, such abbreviations have severe negative impact on program comprehension and IR-based software maintenance activities, e.g., concept location, clustering, and recovery of traceability links [4]. First, maintainers have to guess the full expansion of such abbreviations before they can fully and correctly understand the meaning of software entities [5]. However, the guess may be incorrect, leading to misunderstanding of software entities [6]. Second, IR-based techniques have been widely employed to assist software development and software maintenances, e.g., searching for reusable source code, retrieving source code related to given textual requirements or bug description, and clustering software entities. Although such IR-based techniques have been proved powerful, abbreviations within source code hinders them from reaching their maximum potential [7].

To minimize the negative impact of abbreviations, a number of approaches have been proposed to expand abbreviation in source code [4], [8], [9], [10]. Such approaches can expand a significant percentage of abbreviations accurately. The key of such approaches is to match abbreviations against various dictionaries, e.g., generic English dictionary and dictionaries of common abbreviations. For example, any dictionary word, e.g., '*parameter*', that could result in '*par*' by removing some of its characters is taken as a potential expansion of abbreviation '*par*'. One of the advantages of such dictionary-based approaches is that they are simple and straightforward. Consequently, they are widely used and well recognized. However, one of the disadvantages of such dictionary-based approaches is that there often exist multiple matching dictionary words for a given abbreviation (especially short abbreviations), and it is challenging to tell which one is the correct choice sorely based on dictionaries. The context of the abbreviation, e.g., terms from the enclosing file or enclosing project, has been successfully employed to

- *Y. Jiang, H. Liu, and J. Zhu are with the School of Computer Science and Technology, Beijinag Institute of Technology, Beijing 100081, P.R. China. E-mail: {jiangyanjie, liuhui08, zhujiaqi}@bit.edu.cn.*
- *L. Zhang is with the Key Laboratory of High Confidence Software Technologies, Ministry of Education, Institute of Software, School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, P.R. China. E-mail: zhanglu@sei.pku.edu.cn.*

Manuscript received 11 Feb. 2018; revised 19 June 2018; accepted 22 Aug. 2018. Date of publication 5 Sept. 2018; date of current version 16 July 2020. (Corresponding author: Hui Liu.)
Recommended for acceptance by M. Mezini.
Digital Object Identifier no. 10.1109/TSE.2018.2868762

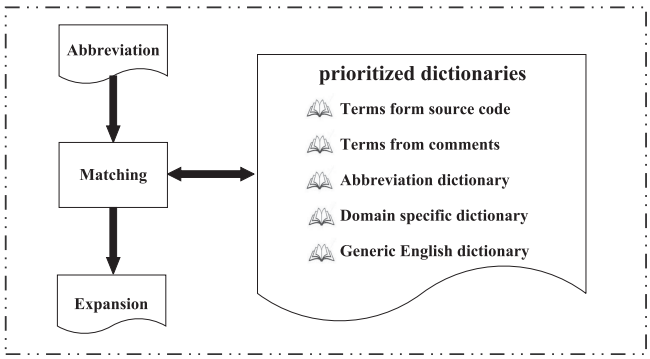

Fig. 1. Typical process of abbreviation expansion.

prioritize such matching words [10], [11], which greatly improves the accuracy of abbreviation expansion. However, such broad and general context often fail to contain the full expansion, or contain multiple potential expansions (matching words), which has severe negative impact on the performance of abbreviation expansion.

To this end, in this paper we propose an automatic approach to improve the accuracy of abbreviation expansion by exploiting the specific and fine-grained context of abbreviations. We notice that the recent empirical study on parameters suggest that actual parameters are often lexically similar to their corresponding formal parameters [12]. Consequently, it is likely that an abbreviation in a formal parameter can find its full expansion in the corresponding actual parameter, and vice versa. We also notice that a significant percentage (36.74 percent) of the abbreviations come from identifiers that are used as either actual parameters or formal parameters (details are presented in Section 4.4.1). To expand such abbreviations, we propose a serial of heuristics to look for their full terms from their corresponding actual (or formal) parameters. We also search for full terms in the data type of the enclosing parameter with a sequence of heuristics. Only if all such heuristics fail, the approach turns to the traditional abbreviation dictionaries. The proposed approach exploits the context of abbreviations as well as the lexical similarity between parameters. We evaluate the proposed approach on seven well known open-source projects. Evaluation results suggest that when only parameter abbreviations are involved, the proposed approach can improve the precision from 26 to 95 percent and recall from 26 to 65 percent compared against the state-of-the-art general purpose approach. Consequently, the proposed approach could be employed as a useful supplement to existing approaches to expand parameter abbreviations.

The paper makes the following contributions:

1) An automatic approach to expanding abbreviations in parameters. The proposed approach differs from existing approaches in that it exploits the specific and fine-grained context of abbreviations with simple and straightforward static syntax analysis. To the best of our knowledge, we are also the first to expand abbreviations by exploiting the lexical similarity between actual and formal parameters.
2) An empirical evaluation of the proposed approach whose results suggest that the proposed approach is accurate.

The rest of the paper is structured as follows. Section 2 presents a short review of related research. Section 3 proposes the approach to expand abbreviation by exploiting the lexical similarity between formal and actual parameters. Section 4 presents an evaluation of the proposed approach on well known open-source projects. Section 5 provides conclusions and potential future work.

2 Related Work

2.1 Abbreviation Expansion

Abbreviation expansion is to expand abbreviations into dictionary words. Fig. 1 presents the typical process of the expansion. It takes a given abbreviation as input, and looks up matching terms (full terms) from a sequence of prioritized dictionaries. Different expansion approaches differ from each other in two aspects: matching algorithms and employed dictionaries.

Abbreviation dictionaries are frequently employed in abbreviation expansion [13]. An abbreviation dictionary contains a list of well known abbreviations as well as their full terms. By looking up at the dictionary, we can retrieve the full terms for a given abbreviation accurately. However, such abbreviation dictionaries are usually constructed manually [14], which is time consuming and tedious. Another disadvantage of abbreviation dictionaries is that they contain only a small part of abbreviations, and thus the matching algorithms may fail frequently to retrieve matching items from such dictionaries [15]. The abbreviation dictionaries miss a large number of abbreviations because of the following reasons. First, they are often manually structured and thus it is challenging for the authors to include abbreviations specific to different domains. Second, the same abbreviation often has different meaning (and thus different full terms) depending on its context. Collecting such abbreviations in the dictionary may confuse its users, and complicates the expansion approaches that depend on it.

Generic English dictionaries are also widely used to expand abbreviations [16]. For a given abbreviation, expansion approaches compare it against each term in the dictionary, and return those that match the abbreviation according to predefined rules (e.g., whether the term begins with the abbreviation). The advantage of looking up expansion from generic English dictionaries is twofold. First, such dictionaries are ready for reuse, and developers do not have to contract such dictionaries again. Second, such dictionaries contain almost all possible English terms, and thus in most cases we can find matching terms for given abbreviations. However, it is quite often that for a given abbreviation there are a large number of matching terms from generic English dictionary. It remains challenging to choose the correct one from such a large number of matching terms. To solve this problem, researchers suggest looking for full terms from those dictionaries first that are constructed according to the context or domain of the abbreviations. Since comments often explain the semantics of source code, it is likely that we can find the full terms of abbreviations from the comments associated with the source code where the abbreviations appear. Consequently, Martino et al. [10], Lawrie et al. [5] and Guerrouj et al. [8] suggest looking for full terms from comments of source code. However,

developers rarely write comments, which significantly reduces the changes to find full terms in comments. Besides that, code comments are often short, and thus may fail to contain most of the full terms. Assuming that developers may use both abbreviations and their full terms within the same project. Cap et al. [14] and Carvalho et al. [11] suggest expanding abbreviations by looking for full terms from the source code. Lawrie et al. [7] try to match a given abbreviation against full terms from the method where the abbreviation appears. In contrast, the approaches proposed by Hill et al. [17] and Corazza et al. [10] are more complex. They match the abbreviation against full terms from its enclosing method, the enclosing class, and the enclosing project in order. The matching algorithms terminate once a matching full term is retrieved. The major difference between them is that they employ different matching algorithms. Hill et al. [17] retrieve matching words with regular expressions whereas $Linsen$ (the approach proposed by Corazza er al. [10]) takes advantage of an approximate string matching technique. Evaluation results suggest that $Linsen$ improves the state-of-the-art [10].

Employing all such dictionaries (e.g., terms from source code and terms from comments) has greatly improved the precision and recall of abbreviation expansion. However, such dictionaries (e.g., dictionaries composed of terms from the whole project) cover a broad context, which reduces the accuracy of abbreviation expansion. The proposed approach differs from existing approaches in that it exploits the specific and fine-grained context (i.e., the data type of the enclosing parameter and the corresponding parameter name) of the abbreviation with simple and straightforward static syntax analysis. To the best of our knowledge, we are also the first to expand abbreviations by exploiting the lexical similarity between actual and formal parameters. The proposed approach can take full advantage of the specific and fine-grained context because it focuses on a special category of abbreviations only (abbreviations in parameters).

Different matching algorithms are used to find full terms from dictionaries. The simplest matching algorithm is looking up full terms directly from the dictionary [5]. A term t from dictionaries is taken as a potential expansion of the given abbreviation *ab* if *ab* is a subsequence of t [18]. If there is more than one matching term, further strategies should be employed to decide which one is correct. Lawrie et al. [4] make suggestions only if the given abbreviation has a single matching term. In other words, it simply ignores the cases where multiple matching terms are retrieved. In contrast, Hill et al. [17] and Carvalho et al. [11] recommend the term with the highest frequency if there are multiple matching terms. In other words, frequently used terms have higher priority. Lawrie et al. [7] choose the one who has the highest lexical similarity with the given abbreviation. Graph-based matching algorithms are proposed by Guerrouj et al. [9] and Corazza et al. [10] to look up potential expansions. They transform terms from dictionaries as a graph where nodes represent letters and edges represent transformation costs. Based on the graph, they search for the shortest path (from the initial letter of the abbreviation to anyone of its matching terms) using Dijkstra algorithm. The one with the shortest path is recommended as the most likely expansion.

2.2 Segmentation of Identifiers

Segmentation of identifiers is to decompose identifiers into sequences of soft words [19]. It is closely related to the expansion of abbreviations because the latter may significantly influence the result of the segmentation (and vice versa).

Segmentation of identifiers could be quite easy if such identifiers follow some common naming conventions, e.g., Camel Case naming convention. In this case, we can segment identifiers according to predefined division markers, e.g., underscores and capital characters [20]. Although such naming conventions are well recognized and are very popular, there are some identifiers (e.g., *arraycopy*) that do not follow such conventions. For such identifiers, the segmentation is much more challenging. To this end, a number of complex and powerful approaches have been proposed to segmenting such identifiers.

Feild et al. [15] propose an approach to split identifiers into their constituent parts based on a greedy algorithm. They employ three lists: the first list is composed of dictionary words, the second is composed of well known abbreviations, and the third is composed of stop words. They decompose a given identifier into a sequence of hard words according to predefined division markers, e.g., underscores and capital characters. For each of the resulting hard words, they compare it against terms in the three lists. If it appears in one of the lists, the hard word is taken as a single soft word that does not require further segmentation. Otherwise, they recursively search for the longest prefix (suffix) of the hard work that is on one of the three lists. The resulting prefix (or suffix) is taken as a soft word, and the remaining is resolved recursively in the same way.

Enslen et al. [21] proposed $Samurai$, an automatic approach to splitting identifiers into sequences of terms based on two term frequency tables. The first table, called program-specific frequency table, counts how many times each term appears within the program. The second table, called global frequency table, counts how many times each term appears in the corpus. Based on these tables, $Samurai$ ranks alternative splits of an identifier according to a scoring function:

$$Score(t,p) = pFreq(t) + \frac{gFreq(t)}{log_{10}(aFreq(p))}, \tag{1}$$

where t is the given term, p is the enclosing program, $pFreq(t)$ is the frequency of t within program p, $gFreq(t)$ is the frequency of term t within corpus, and $aFreq(p)$ is the frequency of all terms within program p. $GenTest$ [4] proposed by Lawrie et al. and $IDSplitter$ [11] proposed by Carvalho et al. are similar to $Samurai$. They differ from $Samurai$ in that they employ different scoring functions, $GenTest$ computes scores with a series of metrics (e.g., $number\_of\_words$, and $co\_occurrence$) and $IDSplitter$ simply accounts the length of terms.

Guerrouj et al. [6] propose a splitting approach named $TIDIER$. Inspired by speech recognition techniques, it finds the best splitting with a greedy search algorithm based on string edit distance. Latter, they propose an enhanced version, named $TRIS$ [9] that handles identifier splitting as an optimization problem. $TRIS$ represents dictionary words as

a tree, and computes the cost of transforming a dictionary word to a string based on the tree. With such cost, *TRISS* compares different splitting and selects the one with the smallest transformation cost.

Graph-based algorithm is also employed for identifier segmentation. Corazza et al. [10] propose an approach to splitting identifiers with an approximate string matching techniques called Baeaz-Yates and Perleberg (BYP) [22]. Ashish Sureka [23] propose a recursive algorithm that could split same-case identifiers. It split a string (identifier) into two substring (noted as S_{left} and S_{right}, respectively). It searches for the optimal split position according to a complex scoring function. The scoring function estimates how good a split is based on the number of hits of S_{left} on image and web search engine, the number of hits of S_{right}, and the length of the two substrings.

2.3 IR-Based Software Engineering

Recent research has shown that software systems contain important and meaning natural language information [24], [25], [26], [27]. By exploiting such information, information retrieval (IR) technologies can automatically solve many problems that previously require considerable human effort, e.g., searching for reusable source code, recovering traceability between requirements and source code, and clustering software entities. While such IR-based technologies are applied to source code, they often rely heavily on the terms within source code. Segmentation of identifiers and expansion of abbreviations help to extract full terms from source code, which facilitates the application of IR techniques on source code.

2.3.1 Recovery of Requirement Traceability

Traceability between requirements and implementation is the link between requirement documents (e.g., features or use cases) and source code (e.g., classes or methods) that implements the requirements [28]. Such links are important for software maintenance and evolution. However, manually creating and managing such links are challenging. Consequently, a number of approaches have been proposed to recover such links automatically [29]. Information retrieval techniques have been proved effective and efficient in recovering such links [30]. The key idea is to take both requirements and source code as traditional documents, and recover the links by computing their lexical similarity [31]: Links are established if the similarity is greater than a predefined threshold. To compare source code with requirement documents written in natural languages, identifiers should be segmented and abbreviations should be expanded properly [21], [8].

2.3.2 Software Clustering

Software systems are becoming difficult to manage and understand because of their increasing complexity [32]. To efficiently manage and understand complex software systems that contain a large number of software entities, IR based software clustering has been proposed successfully. It decomposes a complex software system into some subsystems. Entities within the same subsystem are highly similar whereas entries in different subsystems are distinct [33], [34].

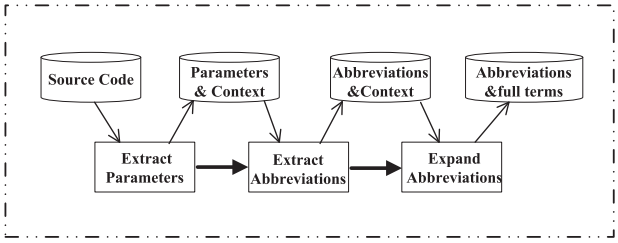

Fig. 2. Overview of the proposed approach.

The key of software clustering is the definition of distance metrics to measure how similar two software entities are. IR-based distance metrics have been proved effective in measure the similarity between software entities. Such IR-based metrics take software entities as traditional documents and calculate their semantic similarity with traditional IR techniques, e.g., latent semantic indexing [35], [36].

2.3.3 Code Search

With the popularity of open-source applications and the power of search engine, code search becomes popular [37]. One of the reasons why developers would like to search for source code on the Internet is that there is plenty of high quality source code on the Internet that developers can reuse for free. Another reason is that by retrieving some example implementation of a given programming task, e.g., reading an Excel document, developers can quickly learn how to implement the task efficiently. There is a number of code search engines in the market, e.g., CodeFinder [38], Codebroker [39], and Google Code Search [40]. The key of these search engines is to match natural language description (users' input) against terms in the source code [38].

3 Approach

In this section, we present an automatic approach to expanding abbreviations in parameter names. An overview of the proposed approach is presented in Section 3.1, and details are presented in the following sections.

3.1 Overview

An overview of the proposed approach is presented in Fig. 2. The input of the approach is the source code of involved application, and the output is a list of abbreviations associated with their full terms. The proposed approach works as follows:

1) It analyzes source code and extracts parameters as well as their context (i.e., parameter types, and their corresponding formal/actual parameters).
2) It segments parameter names, and extracts abbreviations from such names.
3) For each of the resulting abbreviations, it employs a series of heuristics to find its full terms from its context.

Details of the approaches are presented in the following sections.

3.2 Inspiring Dataset for Evidence-Based Approach

To take an evidence-based approach to developing the algorithms and heuristics, we pull out a portion of the dataset in

TABLE 1
Subject Projects

Project	LOC	Number of Functions	Domain	Application/Library	Number of Contributors	Location of Contributors
Retrofit	21,557	1,012	Network	Library	119	USA, Canada, India, Germany
CheckStyle	384,906	17,123	Source Code Analysis	Application	174	Australia,Poland,Germany,USA
PDFsam	33,485	2,107	Office	Application	7	Italy, Holland, Germany, Spain
Bootique	14,600	926	Virtual Machine	Library	10	USA, Belarus, Russia, Belgium
DB-Manager	1,992	182	Database	Application	1	USA
Batik	302,731	23,447	Graphics	Library	5	France, USA, Switzerland
Maven	137,438	8,906	Software Management	Application	67	Canada, Australia, Germany, France
FileBot	40,728	3,116	Multimedia	Application	1	Thailand
Portecle	19,843	3,074	Security	Application	3	Finland, Germany, USA

Section 4, i.e., the first and the second projects in Table 1 (*Retrofit* and *CheckStyle*) for this purpose, and use them to evaluate the ideas behind each heuristics. From each of the projects, we sample 100 parameter abbreviations, and analyze such abbreviations manually. Details of the sampling and analysis are presented in Section 4. For convenience, we call the resulting dataset *inspiring dataset*, and employ it in the following sections to evaluate the ideas behind each heuristic.

3.3 Segmentation of Parameter Names

To extract abbreviations from parameter names, we split parameter names into sequences of soft words. Based on the assumption that such names follow the widely used Camel Case convention [41], we split such names as follows:

1) First, we split a parameter name into a sequence of hard terms according to underscores and numbers. For example, parameter name "*socket_policy3*" is split into $<$'*socket*','*policy*','*3*'$>$.
2) Second, we split a parameter name (or a hard term resulted from the preceding segmentation) into a sequence of terms according to capital letters. Every capital letter is taken as a segmentation flag. An exception is the consecutive capital letters followed by lowercase ones (e.g., "*GETType*"). In this case, only the last capital letter is employed as a segmentation flag. Consequently, "*GETType*" is split into "*GET*" and "*Type*".

For each of the resulting soft words, it is taken as an abbreviation if it does not appear in a generic English dictionary that is composed of full terms only [10]. The following sections explain how such abbreviations are expanded.

3.4 Searching for Expansion

The key part of the proposed approach is to search for the full expansion for a given abbreviation based on a series of heuristics. The input of the search is a given abbreviation $abbr$ from parameter p, its context cts, and a given abbreviation dictionary $Dict$. The context includes the type ($pType$) of the actual (or formal) parameter p, its corresponding formal (or actual) parameter name $pName$:

$$cts(abb) = < pType, pName > . \quad (2)$$

For a given abbreviation, the search strategy is presented in Fig. 3. First, we search for expansion from $pName$. If succeed, we return the expansion, and the search ends. Otherwise, we search for expansion from the data type ($pType$) of the parameter containing the abbreviation to be expanded. Only if we cannot find expansion from $pName$ or $pType$, we look for the expansion from the given abbreviation dictionary $Dict$. If all of the search strategies fail, the proposed approach refuses to make any expansion suggestion for the given abbreviation. The key steps in Fig. 3 are explained in the following sections.

3.5 Searching in Parameter Name

In the *inspiring dataset*, a significant percentage ($32\% = 64/200$) of the abbreviations from formal/actual could find their full expansions in their corresponding actual/form parameter names: 39 out of the 92 abbreviations in formal parameters can find their full terms in their corresponding actual parameter names, and 25 out of the 108 abbreviations in actual parameters can find their full terms in their corresponding formal parameter names. Consequently, for a given abbreviation in formal/actual parameter, we search for its full expansion in the corresponding actual/formal parameter name.

The search is based on a sequence of heuristics, notated as $< H_1, H_2, H_3 >$. It tries each of the heuristics in order, and terminates once any of the heuristics successfully leads to an expansion.

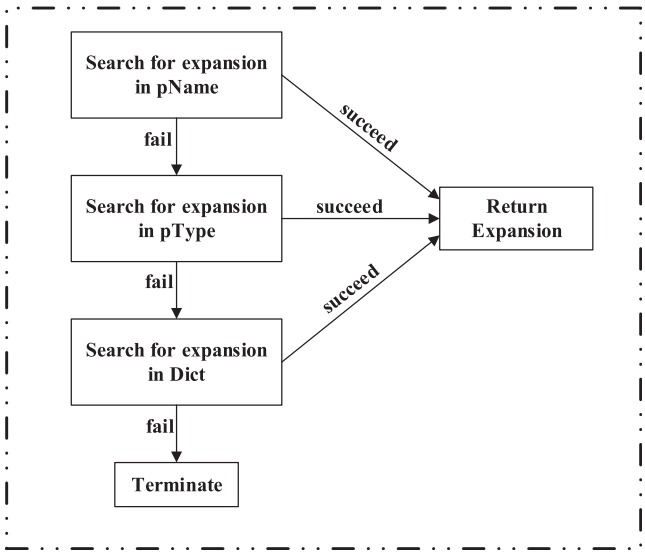

Fig. 3. Searching for expansion.

3.5.1 H_1: Acronym

To shorten the length of an identifier that should have contained a sequence of full terms, developers often concatenate the initial characters of such terms as the abbreviation of the identifier. For example, they may employ "*pn*" to represent "*parameterName*". Among the 64 parameter abbreviations in the *inspiring dataset* whose expansions appear in actual/formal parameter names, 19 are acronyms. To expand such abbreviations, we define a heuristic (noted as H_1) as presented in Algorithm 1.

Algorithm 1. H_1 : Acronym

Input: $abbr$ % abbreviation,
$pName$ % parameter name
Output: $expansion$
1: $terms = segment(pName)$
2: ics = " "
3: **for e***ach* $term$ **in** $terms$ **do**
4: ics = ics + initialCharacter($term$)
5: **end for**
6: **if** $abbr$ equals ics **then**
7: $expansion = terms$
8: **end if**

The input of the algorithm is an abbreviation (abb) from an actual/formal parameter, and its corresponding formal/actual parameter name ($pName$). The parameter name is segmented into a sequence of terms, notated as $terms$ (Line 1). Line 2-5 concatenate the initial character of each term in $terms$, and the resulting string is represented as ics. If the abbreviation is equal to ics, the whole parameter name ($terms$) is returned as the full expansion of abb. It should be noted that the string comparison in the algorithm (as well as other algorithms in this paper) is case insensitive.

The heuristic H_1 works well on the *inspiring dataset*. It successfully expands all of the 19 acronyms whose full expansions could be found in parameter names (i.e., $\text{recall} = 100\%$) whereas it ignores all other parameter abbreviations (i.e., $\text{precision} = 100\%$).

3.5.2 H_2: Prefix

According to our experience, developers often use a short prefix of a long term as the abbreviation. A well known and widely used example is to abbreviate "*string*" as "*str*". Among the 64 parameter abbreviations in the *inspiring dataset* whose expansions appear in actual/formal parameter names, 20 are prefixes. Based on this observation, we propose the second heuristic (H_2) to search for full terms in parameter names. Details of the heuristics are presented in Algorithm 2.

First, we divide the parameter name into a sequence of terms (Line 2). Second, for each of the terms in sequence, we compare it against the abbreviation to validate whether the abbreviation is a prefix of the term. If yes, the term is added to the list $exps$ (Line 3-7) as a potential expansion. From such potential expansions (all elements in $exps$), we select the one with the minimal length as the most likely expansion (Line 8-13).

H_2 works well on the *inspiring dataset*. It successfully expands 20 prefixes whose full expansions could be found in parameter names whereas it ignores all other parameter abbreviations. Among the 20 prefixes, 3 of them result in multiple potential expansions (i.e., $exps$ on Line 9 contains more than one element). For all of them, the strategy of selecting the shortest potential expansion (Lines 9-12) returns the correct expansion.

Algorithm 2. H_2: Prefix

Input: $abbr$ % abbreviation
$pName$ % parameter name
Output: $expansion$
1: $exps \leftarrow \emptyset$
2: $terms = segment(pName)$
3: **for each** $term$ **in** $terms$ **do**
4: **if** $term.beginWith(abbr)$ **then**
5: $exps \leftarrow exps \cup \{term\}$
6: **end if**
7: **end for**
8: $expansion \leftarrow exps[0]$
9: **for each** exp **in** $exps$ **do**
10: **if** LEN$exp <$ LEN$expansion$ **then**
11: $expansion$ = exp
12: **end if**
13: **end for**

3.5.3 H_3: Dropped-Letters

A common way to shorten a long term is to drop some letters of the original term. For example, we may present "*index*" as "*idx*". Such abbreviations are often called dropped-letters [10], [17]. Among the 64 parameter abbreviations in the *inspiring dataset* whose expansions appear in actual/formal parameter names, 25 are dropped-letters. To this end, we propose the third heuristic (H_3) to tell whether a given abbreviation is a dropped-letter abbreviation of any term from the parameter name. The algorithm is presented in Algorithm 3.

Algorithm 3. H_3: Dropped-Letters

Input: $abbr$ % abbreviation
$pName$ %parameter name
Output: $expansion$
1: $exps \leftarrow \emptyset$
2: $terms = segment(pName)$
3: **for each** $term$ **in** $terms$ **do**
4: $i \leftarrow 0$
5: **for** $j = 0; j <$ LEN$term; j++$ **do**
6: **if** $abbr[i] = term[j]$ **then**
7: $i \leftarrow i + 1$
8: **if** $i >$ LEN$abbr$ **then**
9: $exps \leftarrow exps \cup \{term\}$
10: $break$ % skip to the next term
11: **end if**
12: **end if**
13: **end for**
14: **end for**
15: $expansion \leftarrow exps[0]$
16: **for each** exp **in** $exps$ **do**
17: **if** LEN$exp <$ LEN$expansion$ **then**
18: $expansion$ = exp
19: **end if**
20: **end for**

The input, output and the preprocess on $pName$ (Line 2) are the same as those in Algorithm 2. The iteration on Line 3 enumerates each term from the parameter name. Lines 4-13 validate whether the abbreviation $abbr$ could be derived from $term$ by dropping some elements from $term$ without changing the order of the remaining elements. In other words, $abbr$ is a dropped-letter abbreviation of $term$ only if all characters of $abbr$ appear in $term$ in the same order as they appear in abbr. The algorithm adds $term$ as a matching term (Line 9) and ends the validation for $abbr$ and $term$ (Line 10) if all characters of $abbr$ are found successfully in order in $term$ (Line 8). From such potential expansions, we select the one with the minimal length as the most likely expansion (Lines 15-20).

H_3 works well on the *inspiring dataset*. It successfully expands all of the 25 dropped-letters whose full expansions could be found in parameter names. However, we also notice that it results in incorrect expansions for 5 parameter abbreviations, which reduces the precision to $83.3\% = 25/(25+5)$. Among the 25 dropped-letters, 4 of them result in multiple potential expansions (i.e., $exps$ on Line 16 contains more than one element). For all of them, the strategy of selecting the shortest potential expansion (Lines 16-19) returns the correct expansion.

3.6 Searching in Parameter Type

It is quite often that developers may employ an abbreviation of the parameter type as the parameter name. A well known and widely used example is "*catch (Exception e)*" where e represents an instance of type $Exception$. In the *inspiring dataset*, 60 out of the 200 parameter abbreviations can find their full terms in their corresponding parameter types. To expand such abbreviations, we should search for their expansions from their corresponding parameter types. For a given abbreviation $abbr$ and its corresponding parameter type $pType$, the search works as follows:

1) First, we divide $pType$ into a sequence of terms, $terms =< t_1, t_2 ... t_n >$.
2) Second, we concatenate the initial characters of the terms in $terms$, and present the resulting string as ipt. If $abbr$ is equal to ipt, we return $terms$ as the expansion of $abbr$, and the search terminates. The matching algorithm is the same as that in Algorithm 1 (Section 3.5.1) except that it searches in parameter type whereas Algorithm 1 searches in parameter name.
3) We compare $abbr$ against each term in $terms$ to retrieve those terms that begin with $abbr$. The comparison is identical to that in Algorithm 2. If the comparison results in more than one terms, we select the shortest one as the suggested full term of $abbr$.

The search algorithm works well on the *inspiring dataset*. In total it expands 57 abbreviations successfully and no incorrect expansions are returned. However, Heuristic H_3, if applied here to search for dropped-letters in parameter types, would result in low precision (33.3 percent): 2 out of the 3 resulting expansions are incorrect. That is the reason why H_3 is not applied here whereas H_1 and H_2 are applied.

3.7 Searching in Dictionary

Only if all of the heuristics presented in preceding paragraphs fail, the approach turns to the traditional abbreviation dictionary. Such dictionaries, like the one employed by $Linsen$ [10] contain thousands of common abbreviations as well as their full expansions.

Expanding abbreviations by looking up such dictionaries is highly accurate for abbreviations (like "*UK*" and "*COM*") that are well-recognized in different domains. However, many abbreviations do not have the unique interpretation. Instead, they often have different interpretation in different domains. For example, the widely used abbreviation "*USA*" may stand for "*United States of America*", "*United States Army*", or "*United Steelworkers of America*" depending on its context. Consequently, expanding such abbreviation simply by looking up abbreviation dictionaries could be inaccurate. That is the reason why the proposed approach turns to such dictionaries only if all of the heuristics fail.

3.8 Resolution of Conflicting Expansions

For a given abbreviation $abbr$ from an actual/fromal parameter, the proposed approach searches for full terms from its corresponding formal/actual parameter name ($pName$) and the type ($pType$) of its enclosing parameter as suggested in the preceding sections. Overall, the search could be represented as a function:

$$exp = f(abbr, pType, pName). \tag{3}$$

For a given input, i.e., $< abbr, pType, pName >$, the output of the function (the proposed approach) is deterministic because all the heuristics in the preceding sections are deterministic. In other words, the function always generates the same output for the same input. Considering that all heuristics in the preceding section return no more than one expansion for the given input, the search function always results in no more than one expansion.

However, for a given abbreviation $abbr$ from a formal parameter p, the proposed approach may generate more than one expansion. An illustrating example from open-source project (RxJava [42]) is presented in Fig. 4. The proposed approach enumerates all method invocations and tries to expand abbreviations involved in such invocations. Consequently, the proposed approach would try to expand parameter "*s*" for the first time when it encounters the first method invocation (Line 13), and then try to expand it again when it encounters the second and third invocation (Line 18 and Line 23, respectively). For the first invocation (Line 13), the expansion could be represented as:

$$exp_1 = f(``s'', ``Subscriber'', ``stringObserver''). \tag{4}$$

The proposed approach searches for full terms from the corresponding actual parameter name ("stringObserver") first, and heuristic (H_2) in Algorithm 2 finds that "s" is the initial character of the parameter name. Consequently, "string" is returned as the expansion, i.e.,

$$exp_1 = f(``s'', ``Subscriber'', ``stringObserver'') = ``string''. \tag{5}$$

For the second invocation (Line 18), the expansion could be represented as:

$$exp_2 = f(``s'', ``Subscriber'', ``subscriber''). \tag{6}$$

```
1  /* version v2.1.8 */
2  package io.reactivex;
3  public abstract class Flowable<T> implements Publisher<T> {
4    //...
5    public final void subscribe(Subscriber s) {
6    //...
7     }
8  }
9
10 package io.reactivex.internal.operators.flowable;
11 public class FlowableMapTest {
12   //...
13  m.subscribe (stringObserver );
14   //...
15  }
16 public class FlowableMergeTest {
17   //...
18  source.subscribe (subscriber );
19   //...
20 }
21 public class FlowableToSingleTest {
22   //...
23 single.toFlowable().subscribe(subscriber);
24   //...
25 }
```

Fig. 4. Illustrating example from RxJava.

The proposed approach searches for full terms in the corresponding actual parameter name "subscriber". The search algorithm in Section 3.5 returns "subscriber" as the expansion, i.e.,

$$exp_2 = f(``s'', ``Subscriber'', ``subscriber'') = ``subscriber''. \tag{7}$$

For the third invocation (Line 23), the expansion could be represented as:

$$exp_3 = f(``s'', ``Subscriber'', ``subscriber''). \tag{8}$$

The proposed approaches search for full terms in the corresponding actual parameter name "subscriber". The search algorithm in Section 3.5 returns "subscriber" as the expansion, i.e.,

$$exp_3 = f(``s'', ``Subscriber'', ``subscriber'') = ``subscriber''. \tag{9}$$

As a result, the same abbreviation "*s*" from the same parameter have two expansions: "*subscriber*" and "*string*".

To resolve such conflicting expansions, the proposed approach selects among such potential expansions according to their frequency: how many times the given abbreviation is expanded into such expansions. For example, the method $Subscribe(s)$ is called three times, and "*s*" is expanded into "*subscriber*" twice and "*string*" once. Consequently, the proposed approach recommends "*subscriber*" as the full expansion. This algorithm works well on the *inspiring dataset*. For three out of the four parameter abbreviations with conflicting expansions, the proposed approach selects the correct expansions.

It should be noted that abbreviations from different parameters are allowed to have different expansions even if such abbreviations are identical. For example, the parameter "*s*" of "*subscribe(Subscriber s)*" is expanded into "*subscriber*" whereas the parameter "*s*" of "*append(String s)*" is expanded into "*string*". It does not lead to any confliction because the two parameters refer to different software entities although they happen to be lexically identical. In general, if renaming parameters by replacing abbreviations with their expansions leads to confliction, the related expansions are conflicting. Otherwise, the expansions are not conflicting.

3.9 Recursive Expansion

With the search algorithms presented in the preceding sections, the proposed approach generates a list of abbreviations (from parameters) associated with their corresponding expansions. However, such expansions may be abbreviations (instead of full terms) as well. For example, the proposed approach may find that "*s*" is the abbreviation of "*str*". However, the latter ("*str*") is not a full term, neither. It is an abbreviation of "*string*". On the *inspiring dataset*, 14 out of the 124 expansions resulting from the preceding algorithms are not English dictionary words. To guarantee that the suggested expansions are composed of full terms only, we analyze the resulting expansions as shown in Algorithm 4:

1) First, all of the abbreviations as well as their corresponding expansions are stored in a table (noted as t_1). It is noted that the table does not contain those abbreviations that the proposed approach fails to expand on the preceding phases (from Sections 3.5, 3.6, 3.7, and 3.8).
2) Second, for each abbreviation ($abbr$) in t_1, we look up its expansion in a traditional English dictionary that contains full terms only. If the corresponding expansion is not a dictionary term, we move $abbr$ as well as its expansion to a new table (noted as t_2) (Lines 2-6).
3) Third, for each row $r =< abbr, exp >$ in t_2, we search for expansion of exp in t_1 (Line 11). If the selection returns a single expansion (Lines 13-14), we update the row by replacing exp with the resulting expansion and move it back to t_1 (Line 19-21). If multiple expansions are returned, we select the most likely one (as shown in Lines 15-16) as the expansion of exp.
4) Fourth, the third step is repeated until no rows from t_2 could be moved back to t_1.
5) Fifth, for each row $r =< abbr, exp >$ in t_2, we search for expansion of exp in abbreviation dictionaries (Line 30). If succeed (Line 31), we update the row by replacing exp with the resulting expansion (Line 32) and move it back to t_1 (Line 33-34).

Suppose that we have two rows in t_1: $< s, str >$ and $< str, string >$. The first row $< s, str >$ suggests that the abbreviation "*s*" should be expanded into "*str*", and the second row $< str, string >$ suggests that the abbreviation "*str*" should be expanded into "*string*". Algorithm 4 moves row $< s, str >$ from table t_1 to a t_2 (Line 4) because the expansion "*str*" is not in the traditional English dictionary (Line 3). To recursively expand this expansion ("*str*"), Algorithm 4 searches table t_1 to find whether "*str*" has a corresponding expansion in t_1 (Line 11). On table t_1, it finds that "*str*" has been expanded into a full English word "*string*". Consequently, it replaces the row $< s, str >$ with $< s, string >$

(Line 32), and moves the row from t_2 to t_1 (Lines 33-34). The resulting row $< s, string >$ suggests that the abbreviation "*s*" should be finally expanded into "*string*" (instead of "*str*").

Algorithm 4. Recursive Expansion

```
Input: t1   % abbreviations and their initial expansions
Output: t1   % Updated table
 1: t2 ← ∅
 2: for each row in t1 do
 3:     if ( ! InEngDict (row.exp)) then
 4:         move row from t1 to t2
 5:     end if
 6: end for
 7: % The recursive expansion
 8: while true do
 9:     preSize ← size(t2)
10:     for each row in t2 do
11:         expList =select exp from t1 where t1.abbr = t2.exp
12:         newExp ← ""
13:         if size(expList) = 1 then
14:             newExp ← expList[1]
15:         else if size(expList) > 1 then
16:             newExp = select exp from expList group by
    count(exp) desc limit 1
17:         end if
18:         if size(expList) ≠ 0 then
19:             row.exp = newExp
20:             insert row into t1
21:             remove row from t2
22:         end if
23:     end for
24:     if preSize = size(t2) then
25:         break   % the size of Table2 is not changed
26:     end if
27: end while
28: % look up the dictionary
29: for each row in t2 do
30:     newExp ← LookUpAbbrDict (exp)
31:     if newExp ! = null then
32:         row.exp = newExp
33:         insert row into t1
34:         remove row from t2
35:     end if
36: end for
```

4 Evaluation

In this section, we present the evaluation of the proposed approach on seven well known open-source projects.

4.1 Research Questions

The evaluation investigates the following research questions:

- *RQ1*: How often do abbreviations appear in parameters?
- *RQ2*: How often can abbreviations in formal (actual) parameters find their expansions in the data type of their enclosing parameters or in the corresponding actual (formal) parameters?
- *RQ3*: Does the proposed approach outperform the state-of-the-art approach in expanding abbreviation in parameters? If yes, to what extent?
- *RQ4*: How does the abbreviations' length influence the performance of abbreviation expansion?
- *RQ5*: How do the proposed heuristics influence the performance of the proposed approach?
- *RQ6*: Is the proposed approach scalable?

The proposed approach is based on the assumption that a significant part of the abbreviations is contained in identifies that are used as actual or formal parameters. Answering research question *RQ1* helps to validate the assumption. Another assumption taken by the proposed approach is that abbreviations contained in actual/formal parameters names are likely to find their expansions in the data type of the enclosing parameters or in the names of the corresponding formal/actual parameters. Answering research question *RQ2* helps to validate the assumption. Research question *RQ3* concerns the comparison of the proposed approach against the state-of-the-art approaches. We select *Linsen* [10] for comparison because it is accurate and well known, representing the state-of-the-art. As specified in Section 3.7, the proposed approach relies on abbreviation dictionaries. To facilitate the comparison between the proposed approach and *Linsen*, during the evaluation the proposed approach only employs the abbreviation dictionary that is employed by *Linsen* [10]. It should be noted that the purpose of our evaluation is not to validate that the proposed approach is better in general than existing approaches. The only purpose is to validate that better performance could be achieved by focusing on a special and common case of abbreviation expansion and by taking full advance of their specific and accurate context. *RQ4* concerns the impact of abbreviations' length. Answering this question helps to reveal whether the precision and recall are influenced by abbreviations' length. Answering *RQ4* also helps to reveal when the proposed approach performs better (or worse). *RQ5* concerns the impact of different heuristics on the proposed approach. Answering this question helps to reveal why and how the proposed approach achieve high performance. *RQ6* concerns the scalability of the proposed approach, i.e., whether the proposed approach can work on large projects.

4.2 Subject Projects

An overview of the selected nine projects is presented in Table 1. The first column presents the name of projects, and the following columns present the size (LOC and number of functions) of the project, domain, whether it is an application or a library, number of contributors, and the locations of the contributors.

Retrofit[1] is a high quality and efficient HTTP library for type-safe HTTP clients. It encapsulates the underlying code and details, and thus simplifies the work of network operations. *CheckStyle*[2] is a widely used static code analysis tool for Java programs. It supports almost any coding standard (with simple configurations), and automates the checking against given coding standard. *PDFsam*[3] is a powerful project to split and merge PDF files. It is an easy-to-use desktop application with graphical, command line and web

1. https://github.com/square/retrofit
2. https://github.com/checkstyle/checkstyle
3. https://github.com/torakiki/pdfsam

TABLE 2
Abbreviations in Parameters

Projects	All Abbreviations (N_1)	Involved in Actual Parameters (N_2)	Involved in Formal Parameters (N_3)	N_2+N_3	$(N_2+N_3)/N_1$
PDFsam	654	184	61	245	37.46%
Bootique	926	185	111	296	31.97%
DB-Manager	282	77	75	152	53.90%
Batik	13,408	3,702	990	4,692	34.99%
Maven	2,944	747	327	1,074	36.48%
FileBot	1,490	402	210	612	41.07%
Portecle	1,921	739	135	874	45.50%
Total	21,625	6,036	1,909	7,945	36.74%

interfaces. *Bootique*[4] is a minimal Java launcher. It is widely used to build container-less runnable Java applications. *DB-Manager*[5] is a tool to manage and monitor MySQL database services. *Apache Batik*[6] is a toolkit for applications that handle images in the Scalable Vector Graphics (SVG) format, such as viewing, generation or manipulation. It provides developers a set of core modules that can be used together or individually to support specific SVG applications. *Maven*[7] is one of the most popular software project management tools. It is widely employed to manage project's build, reporting and documentation from a central piece of information. *FileBot*[8] is a ultimate tool for renaming the movies, TV shows and downloading subtitles. *Portecle*[9] is a security related applications. It provides user friendly GUI to create, manage and examine keystores, keys, and certificates.

These projects are selected because of the following reasons. First, all of them are open-source projects, which facilitates researchers to repeat the evaluation. Second, they are from different domains, and developed by different programmers. Third, the size of such projects varies dramatically from 1,992 LOC to 384,906 LOC. Such projects cover single-developer small projects as well as large projects where a large number of contributors cooperate extensively. Fourth, the dataset covers libraries as well applications. Finally, developers of such projects are from different countries. Constructing such a comprehensive dataset may improve the generality of the conclusions drawn on the dataset.

4.3 Process

First, for each project we sample 100 abbreviations from parameters. To avoid the bias to the abbreviations of a small number of developers, we draw a stratified sample across developers with Git blame to cover abbreviations from most of the contributors of the projects. As a result, within the same project, the numbers of sampled abbreviations from different developers are roughly equal.

Notably, we exclude identical names, i.e., we do not sample abbreviations from two identical parameter names. If a project contains classes with many overloaded methods, many parameter names will be the same. To avoid the bias to a small number of parameter names, we exclude such identical names and thus the sampled names are different from each other. Notably, for the smallest project *DB-Manager*, the number of abbreviations (48) from unique parameters is smaller than one hundred, and thus we include all such unique parameters for analysis.

Second, three software developers manually expand each of the extracted abbreviations. The three participants are master level students who are familiar with Java. They are not aware of the algorithms used in the approach. According to the information from source code, they recommend an expansion for each of the abbreviations. If they come up with different expansions for the same abbreviation, they discuss with each other. If an agreement is reached, the final agreed expansion is taken. Otherwise, the abbreviation is abandoned. The resulting dataset is publicly available at https://github.com/liuhuigmail/ParameterAbbreviation.

Finally, we evaluate the proposed approach with the resulting dataset. Notably, because the first two projects, i.e., *Retrofit* and *CheckStyle*, have been employed to inspire the evidence-based development of the proposed approach, we exclude the abbreviations from them in the following evaluation. We apply the proposed approach and related approaches to expand the extracted abbreviations. For such automatic approaches, their expansion on a given abbreviation is correct if and only if the resulting full term is identical to that developers manually identified.

The number of abbreviations sampled from each project, as well as the number of involved projects, is a tradeoff between the maximal comprehensiveness of the resulting dataset (abbreviations) and the minimal effort required for the manual analysis on such abbreviations. We can increase the comprehensiveness of the resulting dataset by adding more projects and sampling more abbreviations from each project. However, the bigger the resulting dataset is, the more time-consuming it is to build the golden-set.

We employ precision and recall to evaluate the performance of the approaches. Precision and recall are computed as follows:

$$Precision = \frac{TP}{TP + FP} \tag{10}$$

$$Recall = \frac{TP}{NT}, \tag{11}$$

where TP represents the true positive (correct expansion), FP represents the false positive (incorrect expansion) and NT represents the total number of abbreviations.

4. https://github.com/bootique/bootique
5. https://github.com/debashriroy/DB-Manager
6. https://github.com/apache/batik
7. https://github.com/apache/maven
8. https://github.com/filebot/filebot
9. https://github.com/scop/portecle

TABLE 3
Where to Find Full Terms (For Abbreviations from Formal Parameters)

Project \ Location	Number of Abbreviations (N_1)	In Actual Parameter (N_2)	In Data Type of Formal Parameter (N_3)	$(N_2+N_3)/(N_1)$
PDFsam	52	25	20	87%
Bootique	31	11	13	77%
DB-Manager	16	8	4	75%
Batik	35	17	11	80%
Maven	25	7	13	80%
FileBot	51	20	17	73%
Portecle	26	7	12	73%
Total	236	95	90	78%

4.4 Results

4.4.1 RQ1: Abbreviations in Parameters

To answer research question RQ1, we count all abbreviations in each project as well as those from actual parameters and formal parameters. If a software entity (variable, method, or field) is used somewhere as an actual parameter, all abbreviations in the entity are said to be *involved in actual parameters*. The same is true for formal parameters.

Evaluation results are presented in Table 2. From the table, we observe that a significant part of the abbreviations is from parameters. On average, they account for 36.74 percent of all abbreviations. We also observe that the absolute number of such abbreviations is great, varying from 152 to 4,692 on a single project. On average, each of the projects contains 1,135 abbreviations that are involved in parameters.

From the preceding analysis, we conclude that projects often contain a large number of abbreviations in parameters. Consequently, highly accurate expansion approaches could be valuable even if they are confined to abbreviations from parameters only.

4.4.2 RQ2: Where are the Full Terms

To answer research question RQ2, we manually expand 648 abbreviations from parameters, and investigate how often their full expansions appear in the enclosing parameter type or the corresponding actual/formal parameter names. Results are presented in Tables 3 and 4. The structure of the two tables are almost identical. The first column presents the projects. The second column presents the number of abbreviations from actual (formal) parameters (noted as N_1). The third column presents the number of abbreviations whose full expansions appear in the corresponding actual/formal parameter names (noted as N_2). The fourth column presents the number of abbreviations whose expansions appear in the related parameter types (noted as N_3). The last column presents how often abbreviations from parameters can find their full terms in the corresponding actual/formal parameter names or their enclosing parameter types. We also present the summary in Fig. 5.

From the tables and the figure, we make the following observations:

- First, for a significant part ($28\% = 180/648$) of the abbreviations from actual/formal parameters, their full expansions could be found in their corresponding formal/acteual parameter names. Among the 412 abbreviations from actual parameters, $21\%(= 85/412)$ can find their expansions from the corresponding formal parameter names. Similarly, $40\%(= 95/236)$ of the 236 abbreviations from formal parameters can find their expansions from the corresponding actual parameter names. In total, $28\%(= 180/648)$ of the 648 abbreviations from formal/actual parameters can find their expansions from their corresponding actual/formal parameter names.
- Second, for a significant part ($37\% = 240/648$) of the abbreviations from parameters, their full expansions could be found in the data type of their enclosing parameters.
- Third, for $35\%(= 228/648)$ of the parameter abbreviations, their full expansions could not be found in their corresponding parameter types or parameter names. We represent them as *others* in Fig. 5. Among such abbreviations, $82\%(= 187/228)$ could not find their full expansions within the enclosing projects. A typical example is popular (domain specific or not) abbreviations, e.g., "*URL*", "*Hex*", and "*pwd*". We also notice that for $18\%(= 41/228)$ of abbreviations belonging to *others*, their full expansions appear somewhere within

TABLE 4
Where to Find Full Terms (for Abbreviations from Actual Parameters)

Project \ Location	Number of Abbreviations (N_1)	In Formal Parameter (N_2)	In Data Type of Actual Parameter (N_3)	$(N_2+N_3)/(N_1)$
PDFsam	48	3	14	35%
Bootique	69	11	32	62%
DB-Manager	32	1	15	50%
Batik	65	15	28	66%
Maven	75	29	22	68%
FileBot	49	16	10	53%
Portecle	74	10	29	53%
Total	412	85	150	57%

TABLE 5
Comparison Against State-of-the-Art Approach

Project	The Proposed Approach				Linsen			
	Correct FulTerm	Incorrect FulTerm	Precision	Recall	Correct FulTerm	Incorrect FulTerm	Precision	Recall
PDFsam	70	3	96%	70%	30	70	30%	30%
Bootique	64	4	94%	64%	22	78	22%	22%
Batik	62	3	95%	62%	23	77	23%	23%
Maven	74	5	94%	74%	24	76	24%	24%
DB-Manager	32	1	96%	64%	17	31	35%	35%
FileBot	59	4	94%	59%	23	77	23%	23%
Protecle	59	2	97%	59%	29	71	29%	29%
Total	420	22	95%	65%	168	480	26%	26%

the enclosing projects: 20 ($49\% = 20/41$) in identifiers (but not their corresponding actual/forma parameters) and 21 ($51\% = 21/41$) in comments. Notably, $56\%(= 23/41)$ of such expansions are distributed outside the file where their corresponding abbreviations are found.

Based on the observations we make the conclusion that a significant percentage (up to $65\% = 37\% + 28\%$) of the abbreviations in parameter names can find their expansions from corresponding parameter types or parameter names. We do not have to rely on generic English dictionaries to expand such abbreviations.

4.4.3 RQ3: Compared Against State-of-the-Art-Approach

To answer research question RQ3, we compare the proposed approach against the state-of-the-art approach (*Linsen*). Evaluation results are presented in Table 5. The first column of Table 5 presents the projects. The second to the fifth columns present the results of the proposed approach whereas the sixth to the ninth columns present the results of *Linsen*. The second and the sixth columns present the number of abbreviations that are successfully expanded by the proposed approach and *Linsen*, respectively. The third and the seventh columns present the incorrect expansions made by the proposed approach and *Linsen*, respectively. Notably, the precision and recall of *Linsen* are always the same. *Linsen* expands all abbreviations it encounters, and thus the number of abbreviations (denominator of Formula (11)) is equal to the sum (denominator of Formula (10)) of true positives and false positives. As a result, precision (Formula (11)) is always equal to recall (Formula (10)) for *Linsen*.

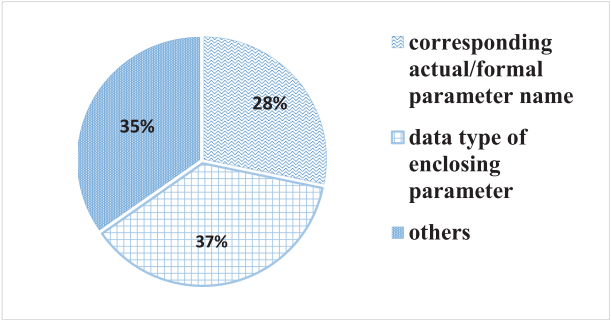

Fig. 5. Where to find full terms.

From the table, we made the following observations:

- First, the proposed approach is highly accurate. Its precision varies from 94 to 97 percent, with an average of 95 percent. In other words, the approach rarely (3 percent chances) makes incorrect suggestions.
- Second, a significant proportion (65 percent) of abbreviations in parameters are expanded successfully by the proposed approach.
- Third, the proposed approach outperforms *Linsen* in expanding abbreviations from parameters. It improves the precision and recall by $265\% = (95\% - 26\%)/26\%$ and $150\% = (65\% - 26\%)/26\%$, respectively. However, it should be noted that the evaluation results do not suggest that the proposed approach is better in general than *Linsen*. The proposed approach is confined to abbreviations in parameters only whereas *Linsen* is more generic and could be applied to all kinds of abbreviations. Consequently, comparing them in general is unfair. The purpose of our evaluation is not to validate that the proposed approach is better in general than existing approaches, but to validate that better performance could be achieved by focusing on a special and common case of abbreviation expansion and by taking full advantage of their specific and accurate context.

From the preceding analysis, we conclude that the proposed approach is highly accurate, and outperforms the state-of-the-art approach in expanding abbreviations in parameters. Consequently, the proposed approach could be employed as an enhancement of existing approaches when the encountered abbreviations are from parameters.

4.4.4 RQ4: Impact of Abbreviations' Length

To investigate how the length of abbreviations affects the performance of abbreviation expansion, we classify abbreviations into subsets according to their length, and apply the proposed approach (and *Linsen* as well) to each subset independently. The evaluation results are present in Table 6 and Fig. 6. The first column of Table 6 presents the length of abbreviations in different subsets where the last subset (4+) includes all abbreviations whose length is no less than four characters. The second column presents the number of abbreviations from each subset. The third to the sixth columns present the performance of the proposed approach whereas the last four columns present the performance of

TABLE 6
Impact of Abbreviations' Length

Length Of Abbreviations	Number Of Abbreviations	The Proposed Approach				Linsen			
		Correct FulTerm	Incorrect FulTerm	Precision	Recall	Correct FulTerm	Incorrect FulTerm	Precision	Recall
1	202	165	10	94%	82%	16	186	8%	8%
2	121	72	6	92%	60%	20	101	17%	17%
3	238	140	4	97%	59%	103	135	43%	43%
4+	87	43	2	96%	49%	29	58	33%	33%
Total	648	420	22	95%	65%	168	480	26%	26%

Linsen. From Table 6 and Fig. 6, we make the following observations.

First, it is challenging for *Linsen* to expand short abbreviations. Its precision (and recall as well) is 8 and 17 percent on expanding one-character and two-character abbreviations, respectively. Short abbreviations are difficult to expand because such abbreviations often lead to a significant number of matching dictionary terms. For example, abbreviation "p" may have hundreds of matching dictionary terms, which makes it extremely challenging to choose the correct one from such a large number of matching terms.

Second, the proposed approach works well for short abbreviations, especially single-character abbreviations. To reveal the reason for such high performance, we manually analyze such abbreviations as well as their expansions. Analysis results suggest that most of such abbreviations are the initial characters of their expansions, e.g., "l" for "letter". The proposed approach can expand such abbreviations successfully because their corresponding parameter names often contain a single term that begins with the given character.

Third, the proposed approach fails to expand a significant part (51 percent) of the long abbreviations that are composed of 4+ characters. The major reason for the failure is that most of such abbreviations should be expanded into sequences of terms although they are not acronyms. A typical example is "*stdout*" from parameter "*stdout*". The camel case based segmentation in Section 3.3 partitions "*stdout*" into one hard words "*stdout*". Since "*stdout*" is not an English dictionary word, the proposed approach takes it as an abbreviation. We note that it is not an acronym of its full expansion ("*standard output*") and thus heuristic H_1 fails. H_2 and H_3 fail as well because both of them assume that the given abbreviation should be expanded into a single dictionary word instead of a sequence of words.

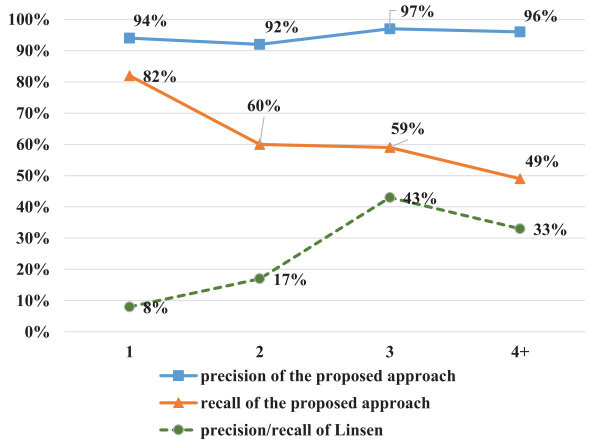

Fig. 6. Impact of abbreviations' length.

Finally, the length of abbreviations has smaller influence on the proposed approach than that on *Linsen*. For the precision and recall of the proposed approach, the coefficient of variation (i.e., the ratio of the standard deviation σ to the mean μ) [43] is 2.03 and 19.28 percent, respectively. In contrast, for *Linsen* it is up to 53.89 percent (both precision and recall), which is significantly greater than that of the proposed approach.

Based on the analysis, we conclude that the length of abbreviations has significant impact on abbreviation expansion. However, its impact on the proposed approach is much smaller than that on *Linsen*.

4.4.5 RQ5: Influence of the Heuristics

The proposed approach contains a sequence of heuristics, i.e., H_1, H_2, and H_3 that are introduced in detail in Sections 3.5.1, 3.5.2, and 3.5.3, respectively. Performance of the used heuristics is presented in Table 7. From this table, we observe that each of the used heuristics expands a large number of parameter abbreviations successfully, which suggests that all of the used heuristics are useful. From this table, we also observe that H_1 and H_2 rarely make incorrect expansions, which suggests that such heuristics are pretty reliable. However, H_3 is significantly more risky, resulting in a precision of 79 percent.

To further investigate the influence of the used heuristics, we repeat the evaluation for several times by disabling the heuristics respectively. For each repeat of the evaluation, we disable one heuristic and others are abled. Each of the heuristics is exactly disabled once. Evaluation results are presented in Fig. 7. From Fig. 7, we make the following observations:

- First, disabling one of the heuristics leads to slight changes in precision. The changes vary from -2 to

TABLE 7
Performance of Used Heuristics

Items	Search in Parameter Names			Search in Parameter Types	
	H_1	H_2	H_3	H_1	H_2
Number of Expansions	85	34	56	119	55
Correct Expansions	82	33	44	117	55
Incorrect Expansions	3	1	12	2	0
Precision	96%	97%	79%	98%	100%

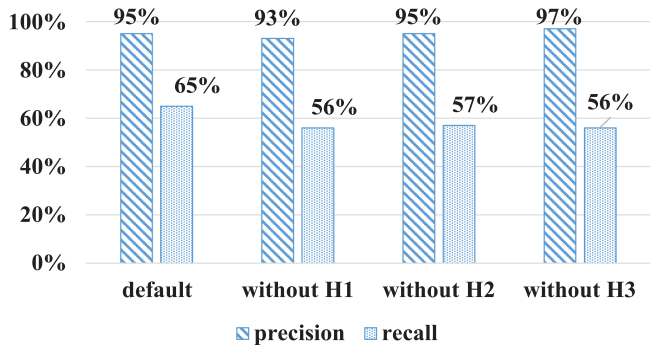

Fig. 7. Influence of used heuristics.

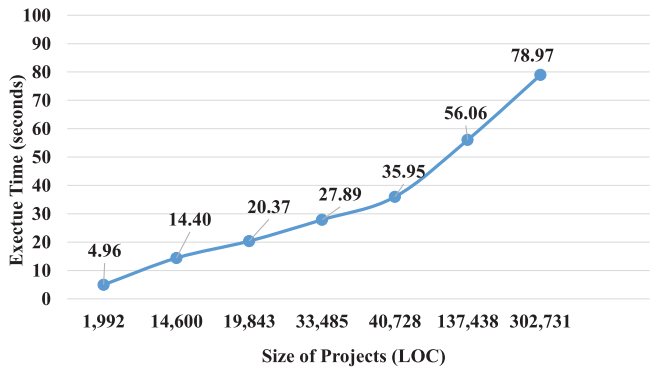

Fig. 8. Scalability of the proposed approach.

2 percent, with an average of 1.3 percent. One of the reasons for the small changes in precision is that all of the heuristics are highly accurate.

- Second, disabling H_1 (acronym) reduces recall significantly from 62 to 56 percent. The evaluation results suggest that a significant part of the abbreviations are expanded by H_1. Similarly, disabling H_2 also reduces (by 8 percent) recall of the proposed approach.
- Finally, disabling H_3 (dropped-letters) increases precision by $2\%(= 97\% - 95\%)$ at the cost of $9\%(= 65\% - 56\%)$ reduction in recall. The results suggest that although H_3 is more risky than H_1 and H_2 (i.e., leading to lower precision), it can expand some abbreviations that H_1 and H_2 cannot expand.

From the analysis, we conclude that disabling any of the heuristics has little impact on precision of the proposed approach but may result in significant reduction in recall (especially H_1).

4.4.6 RQ6: Scalability

We investigate the scalability of the proposed approach by analyzing the relationship between execution time of the proposed approach and the size of projects. The evaluation is conducted on a personal computer with Intel Core i7-6700, 16 GB RAM, and Windows 10. The evaluation results are presented in Fig. 8. The horizontal axis represents the size (LOC) of involved projects, and the vertical axis represents the time that the proposed approach takes to expand all parameter abbreviations (not limited to the manually analyzed sample abbreviations) within a given project.

From Fig. 8, we make the following observations. First, the execution time increases when the size of projects increases. Second, the proposed approach is efficient. It takes less than 1.5 minutes to expand all the 4,692 parameter abbreviations from the large st project (*Batik*) that contains more than 30 thousands of lines of source code. For most of the projects, the execution time is less than one minute.

4.5 Threats to Validity

A threat to construct validity is that we evaluate the suggested expansions against those recommended manually by three students instead of the original developers of the projects. Lack of system knowledge and development experience, they may expand incorrectly, which in turn makes the evaluation of the proposed approach inaccurate. To reduce the threat, abbreviations are excluded from the evaluation if the participants cannot reach an agreement on their expansions.

Another threat to validity of results with respect to manual golden-set creation is that it would heavily weight the results in the favor of the proposed approach if the participants know the proposed approach. In this case, the participants may expand the parameter abbreviations as the proposed approach does. As a result, their expansions are likely to be the same as what the proposed approach suggests, which may seriously bias the evaluation result. To reduce the threat, we exclude the authors of the paper from the manual creation, and recruit students who are not aware of the proposed approach for the manual golden-set creation.

A threat to the external validity is that only 7 projects and 648 abbreviations are involved in the evaluation (besides the 7 projects, 2 projects are employed as evidence to develop the proposed approach). The limited number of involved projects and abbreviations may threaten the generality of the conclusions, i.e., the extent to which such conclusions can be generalized to other situations (other projects and abbreviations). Special characters of such projects may bias the conclusions, e.g., the native language of the developers, the style guide applied by the project, whether the project dose code review, whether it is a single developer project or one with many developers collaborating, and whether it is a library or an application. To reduce the threat, we select such projects that cover diverse characters as shown in Table 1. Evaluation results in Table 5 suggest that the proposed approach is accurate regardless of the diverse characters of such projects. Notably, it is challenging to identify some characters of open-source projects, e.g., the native language of the developers, the style guide applied by the project, and whether the project dose code review. We have sent emails to the contributors to collect such information. However, few of the contributors response. As a result, we fail to find out the style guides applied by such projects, and thus it is not shown in Table 1. We know that some projects, e.g., *PDFsam*, *Batik*, and *Maven*, do code review because we find their code reviews on Github. We also observe that the proposed approach achieves better performance (especially recall) on such projects than other projects. However, for other projects, we do not know whether they do not conduct code review at all or they do it outside Github. Consequently, Table 1 dose not present whether such projects do code review. As a alternative to native languages of developers that we fail to recognize, Table 1 presents the location (countries) of such developers. Evaluation results on Table 5

suggest that the proposed approach works well regardless of the location of developers. To further improve the generality of the conclusions in future, however, evaluation of the proposed approach on more projects and more abbreviations should be conducted.

Another threat to the external validity is that the involved projects may not represent the typical application domains. Including toy applications in the evaluation may reduce the generality of the conclusions. To reduce the threats, we select large and real projects from typical domains for evaluation, e.g., *Retrofit*, *CheckStyle*, *PDFsam*, *Batik*, and *Maven*. However, to investigate the impact of potential factors, e.g., size of projects and the number of contributors, we also include smaller projects for the evaluation, e.g., *DB-Manager*.

5 Conclusions and Future Work

In this paper, we propose an automatic approach to improve the accuracy of abbreviation expansion by exploiting the specific and fine-grained context of abbreviations. Different from exiting dictionary-based approaches, the proposed approach searches for full terms from parameter names and parameter types. We propose a sequence of heuristics to guide the search. We also evaluate the proposed approach on open-source projects. Evaluation results suggest that it is highly accurate. The precision is up to 95 percent, and the recall is 65 percent. Compared to *Linsen*, a state-of-the-art approach, the proposed approach improves the performance significantly in expanding abbreviations in parameters. To the best of our knowledge, we are also the first to exploit the lexical similarity between parameters in abbreviation expansion.

The use of formal and actual parameters is a non-local analysis. When the invoked methods are defined outside the enclosing project (e.g., API methods), local analysis may fail to retrieve the formal parameters of such methods. In this case, analyzing their source code, if available, can retrieve the formal parameter names. However, if such methods are delivered as class files only (and their source code is not available), it is challenging to retrieve the parameter names correctly because class files do not store formal parameter names by default. This may cause negative impact on the practicality of the proposed approach because it depends on formal parameter names. Nevertheless, Java 8 adds an option of embedding formal parameter names in class files [44], which makes such formal parameter names available at the point of analysis.

In future, it would be interesting to combine the proposed approaches with traditional generic approaches. If the proposed approach fail to suggest full terms for a give abbreviation from parameters, we may turn to the traditional generic approaches, e.g., *Linsen*. A potential extension to the proposed approach in future is to take assignments into consideration. According to our experience, the right hand site and left hand site of an assignment often have lexically similar identifiers. Consequently, it likely that for abbreviation on one side of an assignment we may find its full terms on other side of the assignment.

In future, it would be interesting to apply the proposed approach to other statically typed programming languages. Although the prototype implementation of the proposed approach is limited to Java, the approach should be applicable to other programming languages where parameter types and corresponding formal/actual parameter names could be resolved with static analysis. We would like to know how well the proposed approach works for other statically typed programming languages, like *C++* and *Swift*.

Acknowledgments

The authors would like to say thanks to the associate editor and the anonymous reviewers for their insightful comments and constructive suggestions. The work is supported by the National Natural Science Foundation of China (Nos. 61472034, 61772071, 61529201, 61690205), and the National Key Research and Development Program of China(2016YFB1000801).

References

[1] D. Binkley and D. Lawrie, "The impact of vocabulary normalization," *J. Softw.: Evol. Process*, vol. 27, no. 4, pp. 255–273, 2015.
[2] F. Deissenboeck, M. Pizka, and T. Seifert, "Tool support for continuous quality assessment," in *Proc. 13th IEEE Int. Workshop Softw. Technol. Eng. Practice*, 2005, pp. 127–136.
[3] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," in *Proc. 22nd ACM SIGSOFT Int. Symp. Foundations Softw. Eng.*, 2014, pp. 281–293.
[4] D. Lawrie, D. Binkley, and C. Morrell, "Normalizing source code vocabulary," in *Proc. 17th Work. Conf. Reverse Eng.*, 2010, pp. 3–12.
[5] D. Lawrie, H. Feild, and D. Binkley, "Extracting meaning from abbreviated identifiers," in *Proc. Source Code Anal. Manipulation*, 2007, pp. 213–222.
[6] L. Guerrouj, M. Di Penta, G. Antoniol, and Y.-G. Guéhéneuc, "TIDIER: An identifier splitting approach using speech recognition techniques," *J. Softw.: Evol. Process*, vol. 25, no. 6, pp. 575–599, 2013.
[7] D. Lawrie and D. Binkley, "Expanding identifiers to normalize source code vocabulary," in *Proc. 27th IEEE Int. Conf. Softw. Maintenance*, 2011, pp. 113–122.
[8] N. Madani, L. Guerrouj, M. Di Penta, Y.-G. Gueheneuc, and G. Antoniol, "Recognizing words from source code identifiers using speech recognition techniques," in *Proc. IEEE 14th Eur. Conf. Softw. Maintenance Reengineering*, 2010, pp. 68–77.
[9] L. Guerrouj, P. Galinier, Y.-G. Guéhéneuc, G. Antoniol, and M. Di Penta, "TRIS: A fast and accurate identifiers splitting and expansion algorithm," in *Proc. IEEE 19th Work. Conf. Reverse Eng.*, 2012, pp. 103–112.
[10] A. Corazza, S. Di Martino, and V. Maggio, "LINSEN: An efficient approach to split identifiers and expand abbreviations," in *Proc. 28th IEEE Int. Conf. Softw. Maintenance*, 2012, pp. 233–242.
[11] N. R. Carvalho, J. J. Almeida, P. R. Henriques, and M. J. Varanda, "From source code identifiers to natural language terms," *J. Syst. Softw.*, vol. 100, pp. 117–128, 2015.
[12] H. Liu, Q. Liu, C.-A. Staicu, M. Pradel, and Y. Luo, "Nomen est omen: Exploring and exploiting similarities between argument and parameter names," in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng.*, 2016, pp. 1063–1073.
[13] E. Adar, "SaRAD: A simple and robust abbreviation dictionary," *Bioinf.*, vol. 20, no. 4, pp. 527–533, 2004.
[14] B. Caprile and P. Tonella, "Restructuring program identifier names," in *Proc. Int. Conf. Softw. Maintenance*, 2000, pp. 97–107.
[15] H. Feild, D. Binkley, and D. Lawrie, "An empirical comparison of techniques for extracting concept abbreviations from identifiers," in *Proc. IASTED Int. Conf. Softw. Eng. Appl.*, 2006, pp. 365–370.
[16] A. Stevenson, *Oxford Dictionary of English*. New York, NY, USA: Oxford Univ. Press, 2010.
[17] E. Hill, Z. P. Fry, H. Boyd, G. Sridhara, Y. Novikova, L. Pollock, and K. Vijay-Shanker, "AMAP: Automatically mining abbreviation expansions in programs to enhance software maintenance tools," in *Proc. ACM Int. Working Conf. Mining Softw. Repositories*, 2008, pp. 79–88.
[18] A. Apostolico and C. Guerra, "A fast linear space algorithm for computing longest common subsequences," in *Proc. 23rd Annu. Allerton Conf. Commun. Control Comput.*, 1985, pp. 76–84.

[19] C. Caprile and P. Tonella, "Nomen est omen: Analyzing the language of function identifiers," in *Proc. 6th Work. Conf. Reverse Eng.*, 1999, pp. 112–122.
[20] B. Sharif and J. I. Maletic, "An eye tracking study on camelcase and under_score identifier styles," in *Proc. IEEE 18th Int. Conf. Program Comprehension*, 2010, pp. 196–205.
[21] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker, "Mining source code to automatically split identifiers for software analysis," in *Proc. 6th IEEE Int. Work. Conf. Mining Softw. Repositories*, 2009, pp. 71–80.
[22] R. A. Baeza-Yates and C. H. Perleberg, "Fast and practical approximate string matching," in *Proc. Annu. Symp. Combinatorial Pattern Matching*, 1992, pp. 185–192.
[23] A. Sureka, "Source code identifier splitting using yahoo image and web search engine," in *Proc. 1st ACM Int. Workshop Softw. Mining*, 2012, pp. 1–8.
[24] D. Lawrie, H. Feild, and D. Binkley, "Quantifying identifier quality: An analysis of trends," *Empirical Softw. Eng.*, vol. 12, no. 4, pp. 359–388, 2007.
[25] E. Linstead, P. Rigor, S. Bajracharya, C. Lopes, and P. Baldi, "Mining eclipse developer contributions via author-topic models," in *Proc. 4th Int. Workshop Mining Softw. Repositories*, 2007, pp. 30–30.
[26] A. Marcus, D. Poshyvanyk, and R. Ferenc, "Using the conceptual cohesion of classes for fault prediction in object-oriented systems," *IEEE Trans. Softw. Eng.*, vol. 34, no. 2, pp. 287–300, Mar./Apr. 2008.
[27] G. Canfora and L. Cerulo, "Jimpa: An eclipse plug-in for impact analysis," in *Proc. 10th Eur. Conf. Softw. Maintenance Reengineering*, 2006, pp. 2pp–342.
[28] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Trans. Softw. Eng.*, vol. 28, no. 10, pp. 970–983, Oct. 2002.
[29] M. Eaddy, A. V. Aho, G. Antoniol, and Y.-G. Guéhéneuc, "Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis," in *Proc. 16th IEEE Int. Conf. Program Comprehension*, 2008, pp. 53–62.
[30] D. Lucia et al., "Information retrieval models for recovering traceability links between code and documentation," in *Proc. IEEE Int. Conf. Softw. Maintenance*, 2000, pp. 40–49.
[31] A. De Lucia, F. Fasano, R. Oliveto, and G. Tortora, "Can information retrieval techniques effectively support traceability link recovery?" in *Proc. 14th IEEE Int. Conf. Program Comprehension*, 2006, pp. 307–316.
[32] N. R. Jennings, "An agent-based approach for building complex software systems," *Commun. ACM*, vol. 44, no. 4, pp. 35–41, 2001.
[33] A. Marcus and J. I. Maletic, "Identification of high-level concept clones in source code," in *Proc. 16th Annu. Int. Conf. Automated Softw. Eng.*, 2001, pp. 107–114.
[34] M. Shtern and V. Tzerpos, "Clustering methodologies for software engineering," *Advances Softw. Eng.*, vol. 2012, 2012, Art. no. 1.
[35] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *J. Amer. Society Inf. Sci.*, vol. 41, no. 6, 1990, Art. no. 391.
[36] J. I. Maletic and A. Marcus, "Supporting program comprehension using semantic and structural information," in *Proc. 23rd Int. Conf. Softw. Eng.*, 2001, pp. 103–112.
[37] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun, "Repairing programs with semantic code search (t)," in *Proc. 30th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2015, pp. 295–306.
[38] S. Henninger, "Codefinder: A tool for locating software objects for reuse," in *Proc. Automating Softw. Des.: Interactive Des. Workshop Notes*, 1991, pp. 40–47.
[39] Y. Ye and G. Fischer, "Information delivery in support of learning reusable software components on demand," in *Proc. 7th ACM Int. Conf. Intell. user Interfaces*, 2002, pp. 159–166.
[40] (2017). [Online]. Available: http://www.google.com/codesearch
[41] D. Binkley, M. Davis, D. Lawrie, and C. Morrell, "To camelcase or under_score," in *Proc. IEEE 17th Int. Conf. Program Comprehension*, 2009, pp. 158–167.
[42] (2017). [Online]. Available: https://github.com/ReactiveX/RxJava
[43] H. Abdi, "Coefficient of variation," *Encyclopedia Res. Design*, vol. 1, pp. 169–171, 2010.
[44] (2017). [Online]. Available: http://www.oracle.com/technetwork/java/javase/documentation/index.html

Yanjie Jiang received BS degree from the College of Information Engineering, Northwest A&F University, in 2017. She is currently working toward a PhD degree in the School of Computer Science and Technology, Beijing Institute of Technology, under the supervision of Dr. Hui Liu. Her current research interests include software refactoring and software quality assessment.

Hui Liu received BS degree in control science from Shandong University, in 2001, the MS degree in computer science from Shanghai University, in 2004, and the PhD degree in computer science from Peking University, in 2008. He is a professor with the School of Computer Science and Technology, Beijing Institute of Technology, China. He was a visiting research fellow in centre for research on evolution, search and testing (CREST) at University College London, United Kingdom. He served on the program committees and organizing committees of prestigious conferences, such ICSME and RE. He is particularly interested in software refactoring, software evolution and software quality. He is also interested in developing practical tools to assist software engineers.

Jiaqi Zhu received BS degree from the College of Information Engineering, Northwest A&F University, in 2017. She is currently working toward a master's degree in the School of Computer Science and Technology, Beijing Institute of Technology, under the supervision of Dr. Hui Liu. Her current research interests include code generation and software evolution.

Lu Zhang received the BSc and PhD degrees in computer science from Peking University, in 1995 and 2000, respectively. He is a professor with the School of Electronics Engineering and Computer Science, Peking University, P.R. China. He was a postdoctoral researcher in Oxford Brookes University and University of Liverpool, United Kingdom. He served on the program committees of many prestigious conferences, such FSE, OOPSLA, ISSTA and ASE. He was a program co-chair of SCAM2008 and a program co-chair of ICSM17. He has been on the editorial boards of Journal of Software Maintenance and Evolution: Research and Practice and Software Testing, Verification and Reliability. His current research interests include software testing and analysis, program comprehension, software maintenance and evolution, software reuse, and program synthesis.