A Bigram-based Inference Model for Retrieving Abbreviated Phrases in Source Code

Abdulrahman Alatawi
Department of Computer Science
Saudi Electronic University
Riyadh, Saudi Arabia
aalatawi@seu.edu.sa

Weifeng Xu
College of Public Affairs
University of Baltimore
Baltimore MD, USA
wxu@ubalt.edu

Dianxiang Xu
Computer Science Department
University of Missouri
Kansas City MO, USA
dxu@umkc.edu

ABSTRACT

Expanding abbreviations in source code to their full meanings is very useful for software maintainers to comprehend the source code. The existing approaches, however, focus on expanding an abbreviation to a single word, i.e., unigram. They do not perform well when dealing with abbreviations of phrases that consist of multiple unigrams. This paper proposes a bigram-based approach for retrieving abbreviated phrases automatically. Key to this approach is a bigram-based inference model for choosing the best phrase from all candidates. It utilizes the statistical properties of unigrams and bigrams as prior knowledge and a bigram language model for estimating the likelihood of each candidate phrase of a given abbreviation. We have applied the bigram-based approach to 100 phrase abbreviations, randomly selected from eight open source projects. The experiment results show that it has correctly retrieved 78% of the abbreviations by using the unigram and bigram properties of a source code repository. This is 9% more accurate than the unigram-based approach and much better than other existing approaches. The bigram-based approach is also less biased towards specific phrase sizes than the unigram-based approach.

CCS CONCEPTS

• **Software and its engineering** → *Software maintenance tools.*

KEYWORDS

Abbreviation Expansion, Language Model, Software Comprehension, Software Maintenance.

ACM Reference Format:
Abdulrahman Alatawi, Weifeng Xu, and Dianxiang Xu. 2020. A Bigram-based Inference Model for Retrieving Abbreviated Phrases in Source Code. In *Evaluation and Assessment in Software Engineering (EASE 2020), April 15–17, 2020, Trondheim, Norway*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3383219.3383221

EASE 2020, April 15–17, 2020, Trondheim, Norway

ACM ISBN 978-1-4503-7731-7/20/04…$15.00
https://doi.org/10.1145/3383219.3383221

1 INTRODUCTION

Many software applications are written in high-level programming languages, such as Java, C++, and Python. During the implementation phase of these applications, developers have to choose names for identifying programming entities, such as classes, methods, and variables. It is a common practice to use abbreviations to name identifiers for various reasons, such as naming conventions, personal preferences, and laziness of human programmers. For example, the abbreviation "str" may be used to denote "**str**ing" and the character "b" to represent the commonly used word "**b**uffer". Such abbreviations often confuse other software developers and maintainers during the maintenance phase because of their ambiguities – an abbreviation can be interpreted differently by different people. For example, "str" may represent "**str**ing" or "**str**uct", and "b" may refer to "**b**uffer" or "**b**utton". Misinterpretations of abbreviated identifiers not only decrease the productivity of software maintainers but also increase the likelihood of introducing defects. In a mission critical system, they can lead to a code malfunction that causes serious injury, death, or environmental damage [1][2].

Several approaches have been proposed to automatically expand abbreviations to their full meanings that reflect the developer's original intent, including AMAP[3], LINSEN [4], and the unigram-based approach [5][6]. AMAP retrieves the original word for an abbreviation from the contextual information of source code using a list of abbreviating patterns. LINSEN uses an approximate string matching algorithm [7]. The unigram-based approach utilizes the Bayesian probabilistic model and the unigram language model to find the word of the most evidence from a list of candidates. AMAP and LINSEN only focus on expanding an abbreviation to a single word, i.e., unigram. They do not perform well when dealing with abbreviations named after phrases that are composed of multiple unigrams. For example, they have failed to expand "etoa", which stands for a phrase with four unigrams: "**e**stimated", "**t**ime", "**of**", and "**a**rrival". The unigram-based approach overlooks the relationships between unigrams in phrases, which may result in incorrect expansion.

This paper presents comprehensive empirical studies for retrieving abbreviated phrases automatically based on the position paper[8]. The empirical studies focus on investigating the difference between natural and program languages in terms of language comprehension. The studies answer two research questions: (1) How significant is the difference between using a source code repository and using a natural language repository in the bigram-based inference model? (2) Does the bigram-based approach have a bias towards a specific phrase size? The overall approach exploits a bigram-based inference model to choose the best phrase from all

possible candidates for a given abbreviation. It utilizes the statistical properties of unigram and bigrams as prior knowledge and a bigram language model to estimate the likelihood of each candidate phrase. Unigram and bigram originate from the N-gram concept in the field of natural language processing [9]. An N-gram refers to a contiguous sequence of words where *N* is the number of words. An N-gram is a unigram if *N=1* or bigram if *N=2*. The bigram language model treats a sentence or phrase as a sequence of bigrams and computes the probability distribution over the bigrams.

In principle, the bigram-based inference model has advantages over the unigram-based approach because it takes three pieces of evidence into consideration when evaluating a phrase candidate: (1) the properties of each unigram in the phrase, (2) the conditional probability between any two contiguous unigrams in the phrase, and (3) the prior distribution of the abbreviation. To evaluate the bigram-based approach, we have applied it to 100 phrase abbreviations randomly selected from eight open source projects. The experiment results show that it has correctly retrieved 78% of the abbreviations by using the unigram and bigram properties of a source code repository that is obtained from 0.7 million open source software projects hosted on GitHub [6]. This is 9% more accurate than the unigram-based approach and much better than AMAP and LINSEN. The bigram-based approach is also less biased towards specific phrase sizes.

The rest of the paper is organized as follows: Section II presents an overview of the proposed bigram-based approach. Section III describes the bigram-based inference model. Section IV presents our empirical study. Section V reviews the related work. Section VI concludes this paper.

2 THE BIGRAM-BASED APPROACH

2.1 Overview

In this paper, an abbreviation is defined as a sequence of string segments that implies an abbreviated phrase in source code. It contains two or more unigrams that represent the programmer's original intent or domain knowledge about the programming task. For example, the method name "pAFL" in the following code snippet (line 3) is a sequence of four string segments "**p**", "**a**", "**f**", and "**l**". It indicates what the developer intended to do, i.e., "**p**rint **a**pplications **f**orms **l**ist". We refer to this phrase as an *abbreviated phrase* of "pAFL". Similarly, the method parameter "afl" consists of three string segments "**a**", "**f**", and "**l**", which implies what the programmer intended to represent, i.e., an "**a**pplications **f**orms **l**ist". We will use "afl" together with the source code as a running example throughout the paper.

```
1 //Print list of affiliated faculty members
2 // forms from applications forms list
3 void pAFL(ArraryList<Member> afl){
4    for(Member mem : afl){
5        println(mem)
6    }
7 }
```

The bigram-based approach mainly consists of three steps as shown in Figure 1:

(1) Segmenting abbreviation. It partitions a given abbreviation into a set of segment sequences. Each segment sequence is a sequence of string segments. For example, Column 3 of Table 1 lists all segment sequences of "afl".
(2) Generating abbreviated phrase candidates for each segment sequence. For example, for a given segment sequence <"a", "f", "l"> of size three, there are a number of possible abbreviated phrases, such as "**a**pplications **f**orms **l**ist" and "**a**ffiliated "**f**aculty **l**ist". We use "angle brackets" < > to represent a sequence, where segments are separated by commas.
(3) Choosing the best phrase from the abbreviated phrase candidates. It is based on the bigram-based inference model for estimating the likelihood of each candidate phrase. The phrase with the maximum likelihood is considered to be the one that best reflects the developer's original intent.

2.2 Segmenting the abbreviation

Usually, abbreviations are given in a compact form without any dividers. We cannot simply define the size of an abbreviated phrase as the number of characters in the abbreviation. For example, "afl" could represent phrases with different sizes, such as "**af**fi**l**iated" with a size of one, "**af**filiated **l**ist" with a size of two, and "**a**pplications **f**orms **l**ist" with a size of three. In this paper, abbreviation segmentation is to partition a given abbreviation into a sequence of substrings, i.e., segments. Each segment is an abbreviation that represents a unigram. For example, the 3-SS <"a", "f","l"> consists of three segments; each is an abbreviation of a single word. We refer to the size of an abbreviated phrase as the number of segments in a segment sequence.

The algorithm for generating the set of segment sequences for a given abbreviation is described below. It is similar to that for finding the power set of a given set.

(1) Place a binary bit between each pair of adjacent characters in the abbreviation, where each bit can be either "0" and "1". For example, we have "a[bit 1]f[bit 2]l" for "afl".
(2) Generate a string set with all possible bit states. The first column in Table 1 shows the bit states for "afl".
(3) Remove "0" and replace "1" with a marker, i.e., comma, in the abbreviation to split the abbreviation. The third column in Table 1 lists the segment sequences for "afl": {<"afl">, <"af", "l"> , <"a", "fl"> , <"a", "f", "l">}.

Table 1: Generation of Segment Sequences of "afl"

a**[bit 1]**f**[bit 2]**l	**Method**	**Segment Sequence**	**Type**
"a[0]f[0]l"	No partitioning	<"afl">	1-SS
"a[0]f[1]l"	Partition at the second bit	<"af","l">	2-SS
"a[1]f[0]l"	Partition at the first bit	<"a","fl">	2-SS
"a[1]f[1]l"	Partition at all bits	<"a","f","l">	3-SS

For example, the segmentation of "afl" results in the following four segment sequences:

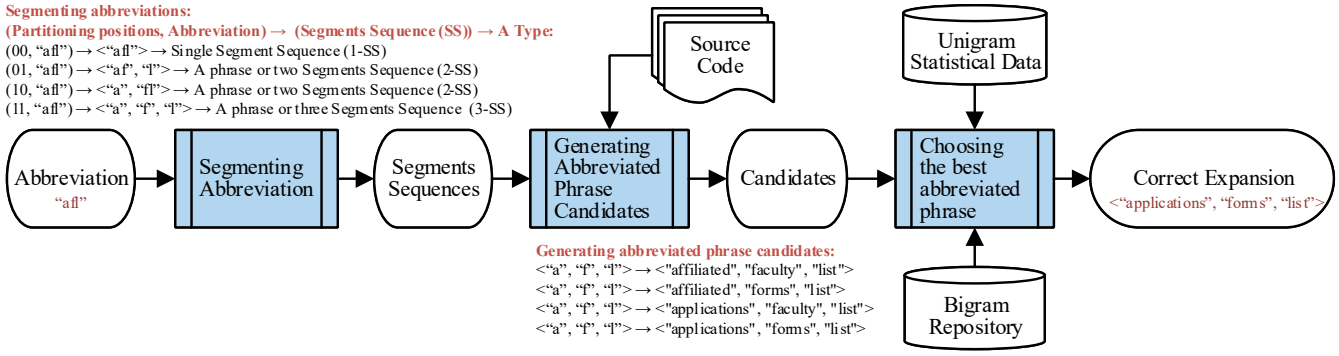

Figure 1: The main steps of the bigram-based approach

(1) A One-Segment Sequence (1-SS): <"afl">. It represents candidate phrases with a size of one, e.g., <"**af**fi**l**iated">,
(2) Two Two-Segment Sequences (2-SS): <"af","l"> and <"a","fl">. They represent candidate phrases with a size of two, e.g., <"affiliated", "**f**acu**l**ty"> and <"**a**pplications", "**f**acu**l**ty">.
(3) A three-Segment Sequences (3-SS): <"a","f","l">. It represents phrase candidates with a size of three, e.g., <"**a**pplications", "**f**orms", "**l**ist">.

2.3 Generating the abbreviated phrase candidates

As mentioned before, each segment in a segment sequence represents the abbreviation of a unigram. Thus, for each segment sequence, we generate the set of its abbreviated phrase candidates as follows: (a) find all unigram candidates of each segment as in the unigram-based approach [5], and (b) transform the segment sequence into the set of all possible phrases, where each phrase is obtained by replacing each segment in the sequence with one unigram candidate.

We apply the search technique in the unigram-based approach [5] to find candidates of single words in the source code. This technique is based on the following statistical patterns between abbreviation segments and abbreviated unigrams:

(1) The position of the first letter of the abbreviation segment and the first letter of a given unigram.
(2) The order of the characters in an abbreviation segment and a given unigram.
(3) The distance between the abbreviation segment and a given unigram in the source code.

Table 2 shows the unigram candidates for all abbreviation segments of "afl". The segment "f", for instance, has two unigram candidates {"**f**aculty", "**f**orms"}.

Generating the abbreviated phrase candidates for a segment sequence is the process of permutating the unigram candidates of all segments. Table 3 shows the four phrase candidates for the 3-SS <"a", "f","l">. The number of phrase candidates is the product of the numbers of unigram candidates of all segments.

Table 2: Unigram candidates for each segment of "afl"

Abbreviation Segments	Unigram Candidates
"afl"	{"**af**fi**l**iated"}
"a"	{"**a**ffiliated", "**a**pplications"}
"f"	{"**f**aculty", "**f**orms"}
"l"	{"**l**ist"}
"af"	{"**af**filiated"}
"fl"	{"**f**acu**l**ty"}

Table 3: Phrase Candidates for sequence <"a", "f","l">

#	Abbreviated Phrase Candidates
1	<"**a**ffiliated", "**f**aculty", "**l**ist">
2	<"**a**ffiliated", "**f**orms", "**l**ist">
3	<"**a**pplications", "**f**aculty", "**l**ist">
4	<"**a**pplications", "**f**orms", "**l**ist">

2.4 Choosing the best phrase

The goal is to choose a phrase from the list of candidates that best reflects the programmer's intent of the abbreviation. It is formulated by Equation 1:

$$\max_{1 \le x \le n} \left[P(c_x|A) \right] \quad (1)$$

where:

- A is an abbreviation
- C is a set of phrase candidates that could be represented by the abbreviation
- n is the size of C, i.e., the number of phrases in the set C
- $c_x \in C$

Equation 1 indicates that (a) for every candidate phrase in the set C for a given abbreviation A, we calculate its likelihood of matching the programmer's intent, and (b) the candidate phrase with the maximum likelihood is considered to be the best one. Therefore, to calculate Equation 1, there are two main problems:

(1) How to find the best phrase from the set of **all phrase candidates** for each segment sequence?
(2) How to find the best phrase from the set of **the best phrases** of all segment sequences of the given abbreviation?

Section III describes our solutions to the above problems.

3 THE BIGRAM-BASED INFERENCE MODEL

Abbreviations are often invented by developers based on their own rules or preferences of naming abbreviations. It can be the first time that other developers or maintainers have ever seen these abbreviations. In this case, it is a challenge for other developers or maintainers to guess the best abbreviated phrase without prior knowledge about how the original programmer created the abbreviations. The proposed bigram-based inference model measures prior knowledge with unigram and bigram properties. It also uses a bigram language model for estimating the likelihood of each candidate phrase.

3.1 Finding the best phrase for a segment sequence

The problem of finding the best phrase for a given segment sequence is formalized by Equation 2:

$$m_{(C_y,S)} = \max_{1 \le x \le n} \left[P(c_x|S) \right] \tag{2}$$

where:

- S is the given segment sequence
- C is a set of phrase candidates generated from S
- n is the size of S, i.e., the number of segments in S
- $c_x \in C$
- $m_{(C_y,S)}$ is the likelihood of c_y, which has the maximum likelihood among all candidates for the given segment sequence S, and c_y is the best candidate, where $1 \le y \le n$

The main task is to determine the conditional probability of c for a given S. To do so, we use a Bayesian inference model expressed by Equation 3. Note that we have dropped the subscript x from Equation 2 as we are evaluating each candidate.

$$P_{bi}(c|S) = \frac{P(S|c) \times P(c)}{P(S)} \tag{3}$$

where:

- $P(c)$ is the probability of the phrase candidate
- $P(S)$ is the probability of the segments sequence
- $P(S|c)$ is prior knowledge of how likely the developer chooses S for given phrase c
- P_{bi} indicates the use of a bigram-based approach

Equation 3 can be transformed into a bigram-based inference model shown in Equation 4. It is a variation of the Bayesian inference model with additional evidence for better estimation: (a) it builds on the prior knowledge (i.e., term ① in Equation 4), and (b) it utilizes a bigram language model (i.e., term ② in Equation 4).

$$\begin{aligned} P_{bi}(< w_1, w_2, ..., w_m > | < s_1, s_2, ..., s_m >) = \\ P(< s_1, s_2, ..., s_m > | < w_1, w_2, ..., w_m >) ① \\ \times P(< w_1, w_2, ..., w_m) ② \\ \times \frac{1}{P(< s_1, s_2, ..., s_m >)} ③ \end{aligned} \tag{4}$$

where:

- $c =< w_1, w_2, ..., w_m >$ and w_i is a unigram of the abbreviated phrase candidate c
- $S =< s_1, s_2, ..., s_m >$ and s_i is a segment in the segment sequence c
- There are existing relationships between w_i and s_i such that w_i is a unigram candidate of segment s_i

3.2 Prior knowledge measurement

The prior knowledge, i.e., term ① in Equation 4, refers to the probability that a developer uses the abbreviation $<s_1, s_2, ..., s_m>$ to represent $<w_1, w_2, ..., w_m>$ (e.g., how likely a developer uses <"a", "f","l"> for <"affiliate", "faculty", "list">). It is determined by Equation 5, which is called the abbreviation representation model for phrases.

$$P(< s_1, s_2, .., s_m > | < w_1, w_2, ..., w_m >) = \prod_{j=1}^{m} P(s_j|w_j) \tag{5}$$

Let us first prove that Equation 5 holds when $m = 2$, i.e., for a phrase with two unigrams. In this case, it reduces to Equation (6):

$$\begin{aligned} P(< s_i, s_j > | < w_i, w_j >) = P(s_i|w_i) \times P(s_j|w_j) \\ where\ i, j \in < 1...n > and\ j = i + 1 \end{aligned} \tag{6}$$

Proof:

$$P(< s_i, s_j > | < w_i, w_j >) = \frac{P(s_i \cap s_j \cap w_i \cap w_j)}{P(w_i \cap w_j)}$$

$$= \frac{P(s_i \cap s_j) \times P(w_i \cap w_j)}{P(w_i) \times P(w_j)} \quad \text{[The conditional probability]}$$

$$= \frac{P(s_i \cap w_i)}{P(w_i)} \times \frac{P(s_j \cap w_j)}{P(w_j)} \quad \text{[Chain rule]}$$

$$= P(s_i|w_i) \times P(s_j|w_j) \quad \text{[Reverse Kolmogorov definition]}$$

Similarly, we can prove Equation 5 by induction based on Equation 6. For example, we can measure how likely a developer uses *"afl"* to represent the phrase *"affiliated faculty list"* as follows:
P(<"a", "f","l">|<"affiliated", "faculty", "list">)
= P("a"|"affiliated")× P("f"|"faculty")×P("l"|"list")

It is not difficult to see that the abbreviation representation model for phrases is essentially a product of $P(s|w)$ for each pair (s, w) in a given segment sequence and its corresponding phrase. $P(s|w)$ can be estimated using the probabilities of three properties described in [6]. These properties describe developers' abbreviation naming strategies using the statistical distribution between the pair (s, w) extracted from 0.7 million open source projects. In the following, we briefly describe these strategies.

Abbreviation Type Choosing (ATC) strategy. There are mainly two abbreviation types when programmers decide to choose abbreviations: Consecutive Characters Abbreviation (CCA) and Nonconsecutive Characters Abbreviation (NCA). CCA uses the first n consecutive characters as the abbreviation for a given English phrase whereas NCA uses n nonconsecutive characters as the abbreviation. Equation 7 shows the likelihood of choosing a segment s for a given unigram w.

$$P_{ATC}(s|w) = \begin{cases} if(w \longmapsto s) = CCA, & 83.93\%. \\ if(w \longmapsto s) = NCA, & 16.09\%. \end{cases} \quad (7)$$

Typing Effort Saving (TES) strategy. TES measures the balance between the "laziness" of developers and the code readability when creating abbreviations. Formally, the TES for a given pair (s, w) is defined by equation 8:

$$TES = 1 - \frac{length(s_j)}{length(c_j)} \quad (8)$$

For example, TES(int, integer)=57.14%, and TES(i, integer)=85.71%. The range of TES is (0, 1), excluding 0 and 1. Zero means that no abbreviation is used. We use the frequency distribution of TES, i.e., $P_{TES}(TES(s, w))$, to measure such balance. A study shows that 21% programmers choose the abbreviation "*i*" for the unigram integer and only 13% choose "*int*". The probability of P_{TES} is found from the TES distribution in figure 2.

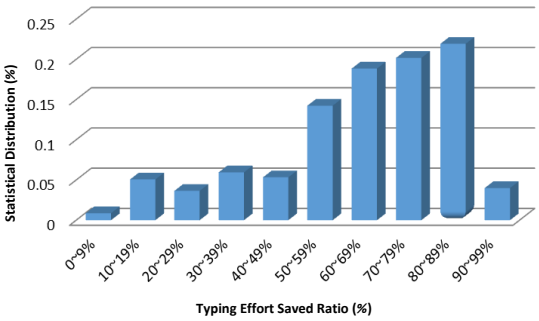

Figure 2: Distribution of Typing Effort Saving Ratio [6]

Context Sensitiveness (CS) strategy. It measures how developers choose abbreviations based on its context. It makes more sense for programmers to choose abbreviations based on the words that appear in its context in the source code. It is uncommon to name an abbreviation that comes from nowhere in the source code. The following formula shows the likelihood of finding the corresponding abbreviation for a given unigram in terms of distance (i.e., line number). A study shows that the probability of Context Sensitiveness for a segment s of a unigram w is given by equation 9:

$$P_{CS}(s|w) = (0.83)e^{(-2.53\times|s-w|)} \quad (9)$$

We use the above strategies to measure prior knowledge. The probability of a segment s of a unigram w is shown in Equation 10.

$$P(s|w) = P_{ATC}(s|w) \times P_{TES}(s|w) \times P_{CS}(s|w) \quad (10)$$

Using the abbreviation representation model for phrases, i.e., Equation 5, we can measure how likely developers use the abbreviation *"afl"* to represent the phrase *"affiliated faculty list"*.

3.3 Bigram language model

The bigram-based inference model takes advantage of a bigram language model to provide additional evidence. Specifically, term ② in Equation 4 is the probability of a phrase calculated by using a bigram language model. Unlike the unigram language model used by [6][5], the probability of each word only depends on that word's own probability in the document. The bigram language model considers the probability of observing the current word in the context history of the preceding word. In principle, it will produce a better estimation result. Equation 11 calculates the probability of a phrase represented by a sequence of words $< w_1, w_2, .., w_n >$.

$$P(< w_1, w_2, .., w_n >) = \prod_{i=1}^{n} P(w_i|w_{i-1}) \quad (11)$$

For example, the probability of the phrase "affiliated faculty list" is calculated as follows: *P("affiliated faculty list"= P("affiliated") × P("faculty"| "affiliated") × P("list"| "faculty").*

3.4 Abbreviation frequency

Term ③ in Equation 4 needs to use the frequency of abbreviations, i.e., $P(< s_1, s_2, ..., s_m >)$. The frequency of each abbreviation is usually pre-calculated for a given repository. There is no direct way to compute the frequency of all abbreviations named after phrases because we do not know whether or not an abbreviation is named after a unigram or a phrase. We propose the following alternative approach to calculate the frequency:

(1) Select a natural language repository or a source code repository.
(2) Count all abbreviations, denoted as *abb_all*. An abbreviation is defined as a string if it is not found in the repository.
(3) Count all abbreviations named after unigrams, denoted as *abb_unigram*, based on the unigram naming patterns [6].
(4) The number of abbreviations named after phrases is $abb\_bigram = abb\_all - abb\_unigram$.

Note that we use the Laplace add-one smoothing method [10][11] to increase the zero probability of an abbreviation to a small positive number. This prevents the denominator of term ③ in Equation 4 from being a zero by counting an abbreviation once if it is the First-Time-Seen (FTS) abbreviation during the phrase retrieving process.

3.5 Normalization

Equation 2 chooses the phrase c_y with the maximum likelihood of matching developer's intent from all phrases C generated from one segment sequence S, such as <"a", "f", "l">. We use the notation $m_{(y,S)}$ to denote the likelihood of such c_y, and y is the index of the phrase in the set C. Note that an abbreviation can be partitioned into multiple segment sequences with different sizes (e.g., Table 3), and these segment sequences produce a set of the best abbreviation phrases. Equation 12 shows how to pick the best phrase from such a set of best abbreviation phrases. We normalize $m_{(y,S)}$ so that long phrases have a better chance to be chosen as the best phrase.

$$\max_{S \in SS} [m_{(y,S)}]^{\frac{1}{|S|}} \quad (12)$$

where:

- SS is a set of segment sequences generated from a given abbreviation
- $|S|$ is the size of S, i.e., the number of segment in S

Table 4 shows the likelihood of each phrase candidate for a segment sequence <"a","f","l"> ranked from high to low. The best

phrase is "**a**pplications **f**orms **l**ist". $P(S)$ is omitted as they are the same for the same segment sequence. Also, P_{ATC}, P_{TES}, and P_{CS} are the same for each pair (s, w) for a given phrase candidate.

4 EMPIRICAL STUDY

The empirical study aims at answering the following research questions:

RQ1: Does the bigram-based approach outperform the existing approaches, including AMAP[3], LINSEN[4], and the unigram-based approach[5]?

RQ2: How significant is the difference between using a source code repository and using a natural language repository in the bigram-based inference model?

RQ3: Does the bigram-based approach have a bias towards a specific phrase size?

We use retrieving accuracy as the main performance indicator. It is defined as the percentage of the total number of correctly retrieved phrases over the total number of abbreviations.

4.1 Subjects

The subjects for the empirical study consist of 100 phrase abbreviations from eight open source projects at GitHub [12]. As shown in Table 5, these projects have varying sizes (in terms of KLOC) and different levels of maturity. They are implemented by varying numbers of developers who may use different abbreviation methods. The same set of projects, as well as the same data pre-processing frameworks and techniques, are used in [5][6] for the unigram extraction and analysis. These unigram properties build a foundation for the bigram-based analysis. The 100 abbreviations are randomly chosen from these projects for evaluating the bigram-based approach. Table 6 shows three examples from the abbreviation set. Each abbreviation has several attributes, including the length of the abbreviation, the location of the abbreviation (i.e., project, class, line number), the manually retrieved phase (i.e., ground truth), and the size of the retrieved phrase. For example, the abbreviation "wc" is located at the third line of classUtils.java file in the project OpenProj. The manually retrieved phrase from the class file is "**w**orking **c**alendar".

4.2 RQ1: Does the bigram-based approach outperform the existing approaches?

We compare the bigram-based approach to three existing approaches, AMAP, LINSEN, and Unigram-based, by using two types of repositories when applicable: (a) **N**atural **L**anguage **R**epository (NLR), which is a collection of one Terabyte of web corpus, such as textbooks and news archives [13], and (b) **S**ource **C**ode **R**epository (SCR), which is obtained from 0.7 million open source software projects hosted on GitHub [6]. When NLR is used, the existing approaches all rely on the Unigram Properties extracted from NLR (denoted as UP-NLR), whereas the proposed bigram-based approach exploits the Bigram Properties extracted from NLR (denoted as BP-NLR). In addition, both the unigram-based and the bigram-based approaches may use SCR. As such, we have conducted four experiments with the same subject abbreviation list: unigram using the unigram properties of NLR, unigram using SCR, bigram using the bigram properties of NLR, and bigram using SCR.

The experiment results are presented in Table 7, where AC and CW stand for Acronym and Combination Word, respectively. Acronyms refer to well-known abbreviations such as "GPS"="**g**lobal **p**ositioning **s**ystem". Combination words use either CCA or NCA abbreviations with an acronym, such as "oid"=="**o**bject **id**entifier". For comparison, Table 7 also includes the performance of AMAP and LINSEN from the literature, which was based on 72 abbreviations and their corresponding phrases. It is worth noting that although AMAP and LINSEN contain 250 abbreviations in their empirical study, only 72 of these abbreviations are named after phases, which fall into two categories: acronym (AC) and combination word (CW). We are only interested in comparing the effectiveness of each algorithm that retrieves these abbreviated phases. The main observations related to RQ1 are as follows:

- Both the bigram-based approach and the unigram-based approach are much better than AMAP and LINSEN. This is because neither AMAP nor LINSEN was designed for retrieving phrases. The bigram-based approach achieved the best overall accuracy (78%) when SCR is used. For example, the abbreviation "fa" number 68 (in Table 8) is a 2-segment sequence abbreviation with two candidates "**f**ormatter **a**rray" and "**f**inal **a**rray". The abbreviation is correctly expanded to the first candidate using SCR because the bigram "**f**ormatter **a**rray" is commonly used (i.e., has a higher frequency) in the source code but less common in the natural language. The second best approach is the unigram-based approach using SCR. The overall accuracy is 69%. The bigram-based approach using SCR outperformed it by 9%.
- Acronyms are easier to be recognized than combination words. The bigram-based approach has achieved 82.26% in AC and 71.05% in CW. The AC type is much easier to be retrieved than the CW type because the bigram-based approach takes the frequencies of all bigrams in each phrase candidate into consideration, whereas the bigram frequencies in the AC type phrases often have higher frequencies than the CW type. For example, the abbreviation "mem" number 87 (in Table 8) can represent two abbreviated phrase candidates, the acronym"**m**otion **e**vent **m**ouse" and the combined words "**m**ouse **e**vent **m**ask". When the bigram-based approach wants to determine which phrase to pick, it will consider the frequency of the bigrams "**m**otion **e**vent" and "**e**vent **m**ouse" for the phrase "**m**otion **e**vent **m**ouse" as well as the frequency of the bigrams "**m**ouse **e**vent" and "**e**vent **m**ask" for the phrase "**m**ouse **e**vent **m**ask". The approach is more likely to pick an acronym because its bigrams have higher frequencies.

4.3 RQ2: How significant is the difference between using a source code repository and using a natural language repository in the bigram-based inference model?

The N-gram property is a dimension of prior knowledge related to a specific language. Therefore, our hypothesis is that using SCR will result in a better performance than using NLR in the bigram-based approach. The results in Table 7 show that this hypothesis is

Table 4: Best phrase candidate for the sequence <"a","f","l"> is ranked at the top

#	Abbreviated Phrase Candidates, c	$P(S\|C)$	$P(C)$	$P(S)$	$P_{Bi}(C\|S)$
1	**<applications|forms|list>**	0.01558	5.659e-07	0.002523	**0.01518**
2	<affiliated|faculty|list>	0.006705	6.019e-07	0.002523	0.01169
3	<applications|forms|list>	0.006705	5.659e-07	0.002523	0.01146
4	<affiliated|faculty|list>	0.002885	6.019e-07	0.002523	0.008829

Table 5: Subject Programs

Program	KLOC	Ver.	#Developers	Description
Hibernate	936	5.5	252	Object/Relational Mapper Tool
HSQLDB	338	2.3.4	6	Relational Database Tool
Ant	269	1.9	27	Java Code Testing Tool
Google Guava	220	5	99	Google's Core Library
OpenProj	215	2009	15	Project Management
Junit	40	4.12	133	Java Code Testing Tool
Checkstyle	38	7.2	9	Eclipse Check Style Plugin
Libsdl-Android	22	1.3	10	Android SDL Library

Table 6: Sample Abbreviations from the Empirical Study

Abbreviation	wc	lce	mmcr
Project	OpenProj	Checkstyle	Libsdl-android
Line Number	63	169	341
Class Name	ClassUtils	Annotation-UseStyle-Check	Main-Activity
Manually Retrieved Phrase	**w**orking **c**alendar	**l**oading **c**ollection **e**ntry	**m**in **m**ax **c**ategory **r**enderer
Size of Phrase	2	3	4

valid. The bigram-based approach using SCR has achieved an overall accuracy of 78%, whereas the bigram-based approach using NLR has an overall accuracy of 69%. The improvement is 9%. Additionally, the hypothesis is also valid for the unigram-based approach. Using SCR has an overall accuracy of 69%, whereas using NLR has an overall accuracy of 68%. For example, the abbreviation "tinum" number 96 (in Table 8) can represent two abbreviated phrase candidates, "**t**ask **i**d **n**umber" and "**t**itle **i**d **n**umber". When the bigram-based approach wants to determine which phrase to pick, it will consider the frequency of the bigrams in the utilized repository. The bigram

Table 7: Comparison of Accuracy

		Total		Correctly Retrieved		Retrieving Accuracy		Overall Retrieving Accuracy
		AC	CW	AC	CW	AC	CW	
AMAP	SCR	49	23	23	4	46.9%	17.4%	37.5%
LINSEN	SCR	49	23	18	15	36.7%	65.2%	45.8%
Unigram Approach	NLR	62	38	45	23	72.58%	60.53%	68.0%
	SCR	62	38	46	23	74.19%	60.53%	69.0%
Bigram Approach	NLR	62	38	46	23	74.19%	60.53%	69.0%
	SCR	62	38	51	27	82.26%	71.05%	78.0%

AC: Acronym, CW: Combination Word

"**t**ask **i**d" is common bigram in SCR whereas the bigram "**t**itle **i**d" is common in NLR, therefore, the bigram-based model pick the phrase with higher frequency as best expansion according to the repository.

4.4 RQ3: Does the bigram-based approach have a bias towards a specific phrase size?

The bias towards a specific phrase size, called "retrieving size bias", means that a retrieving algorithm has unbalanced favors to choose phrases with a particular size given all possible sizes of segment sequences. If an algorithm has no bias, its accuracy rates for all sizes of segment sequences should be similar. A less biased algorithm often shows more consistency and therefore is more reliable when choosing the best phrase from a candidate set. For example, the abbreviation "afl" can be partitioned into segment sequences with a size of two and three. We expect that the retrieving accuracy will be about the same (i.e., 50% for phrases with sizes of two and three) in different source code contexts.

We analyze the issue of size bias as follows:

(1) Apply the unigram and bigram-based approaches to the 100 abbreviations using both NLR and SCR. Thus we have four method groups: UP-NLR (the unigram-based approach with NLR), UP-SCR (the unigram-based approach with SCR), BP-NLR (the bigram-based approach with NLR), and BP-SCR (the bigram-based approach with SCR). The results of abbreviation expansions are shown in Table 8.
(2) Divide each of the four groups into three subgroups in terms of phrase sizes: 2, 3, and 4. Phrase size refers to the size of the segment sequence from which the phrase is extracted.
(3) Calculate the retrieving accuracy for each size subgroup in each method.

(4) Determine the bias, which is the standard deviation of accuracy rates of the three size subgroups in each method.

Figure 3 shows the retrieving accuracy rates of UP-NLR, UP-SCR, BP-NLR, and BP-SCR with respect to phrase sizes. For example, the accuracy rates of BP-SCR are 89.71%, 57.14%, and 45.45% for phrase sizes 2, 3, and 4, respectively. The biases of UP-NLR, UP-SCR, BP-NLR, and BP-SCR are 25.49%, 26.58%, 21.24%, and 22.93%, respectively. This indicates that (a) the bigram-based approach is less biased than the unigram-based approach. It is not surprising because the retrieving accuracy is improved from 27.27% to 45.45% for given 4-segment-sequences due to the normalization shown in Equation 12. The normalization improves the chance of picking phrases with more words rather than penalizing them. For example the abbreviation "mem" (the 87th abbreviation in Table 8) is expanded correctly to the phrase "**m**otion **e**vent **m**ouse" using SCR rather than "**m**ous**e** **m**ask", and (b) Using BP-NLR is 1% less biased than using BP-SCR, however, the difference is not statistically significa

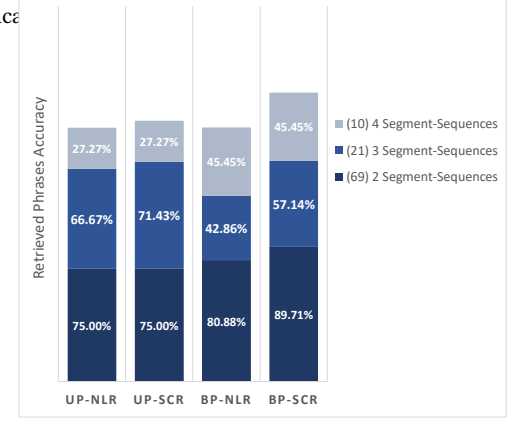

Figure 3: Accuracy rates with respect to phrase sizes

4.5 Threats to Validity

Threats to *internal validity* concern anything that could influence our study results. In our subject data, it is possible that some of the expanded phrases of the 100 abbreviations used as the ground truth could be incorrect because they were manually collected and identified. To limit this threat, we hired two computer science students to verify the correctness of the expanded phrases. In addition, the 100 abbreviations used by the unigram-based and bigram-based approaches are different from the 72 abbreviations used by AMAP and LINSEN. This is because the bigram-based approach relies on the context of abbreviations in the source code as important evidence for guessing the intent of source code in order to find the abbreviated phrases, whereas AMAP and LINSEN do not rely on such context information. Thus, the abbreviation subjects used for evaluating AMAP and LINSEN do not have context information.

Threats to *external validity* concern the possibility of generalizing our results. First, our evaluation dataset contains multigram abbreviations whose sizes range from two to seven alphabetic characters. Therefore, our result may not be generalized to abbreviations that are longer than seven alphabetic characters. Second, the implementation of our approach has excluded a set of reserved words used in programming languages (e.g., "null", "if", and "for"), and conjunction words (e.g., "and", "or", and "since"). More reserved words need to be added when other programming languages are included in SCR. As a consequence, the bigram-based approach does not apply to particular abbreviations with the above excluded words.

Threats to *conclusion validity* concern the relations between the approach and the results. The proposed approach uses N-gram properties as domain knowledge. However, N-gram has its own pitfalls [14]. It is beyond the scope of this paper to address these pitfalls, though.

5 RELATED WORK

Expanding abbreviations to their full words has been investigated by several research groups. In this section, we focus on reviewing and comparing the most relevant approaches, including Lawrie et al. [15], Hill et al. [3], Corazza et al. [4], and Alatawi et al. [5].

Lawrie et al. presented a two-step approach: identifier splitting and abbreviation expanding [15]. The identifiers are split according to word boundaries using such known techniques as CamelCasing or underscore in '*sponge_Bob*". For abbreviation expansion, the potential expansions are extracted from the source code, and then the best expansion is determined by using a phrase finder [16] or Krovetz stemming [17]. The evaluation using a set of 64 abbreviations shows that the above approach only achieves an accuracy rate of 20%.

Hill et al. proposed "AMAP", which is a continuation of the aforementioned approach. While abbreviations are treated in a similar fashion, AMAP searches for potential expansions by using the closest scope (e.g., names, statements, method, comments, program) and five regular expression patterns in order (acronym, prefix, dropped letter, and combination word). AMAP uses a Most Frequent Expansion (MFE) technique to deal with multiple expansions. The evaluation with 250 abbreviations shows that AMAP has an accuracy rate of 58.8%.

"LINSEN" uses a graph representation where each node represents a character in the abbreviation, and each edge represents the approximate matching dictionary word. The edge cost is based on a tolerance function in Baeza Yates and Perlberg (BYP) algorithm [7]. "LINSEN" also uses an abb-deletions on the expanded forms deleting final, vowels, or consonants. In addition, "LINSEN" expands abbreviations according to word frequencies. The evaluation shows that "LINSEN" has an accuracy rate is 59.1%, which is only slightly better than AMAP.

Alatawi et al. used a probabilistic model to expand abbreviations based on Bayesian inference. They first extract a list of potential candidates from the source code of the abbreviation, and estimate the likelihood of each candidate being the correct expansion. They used a unigram-based inference model with two components: (1) prior knowledge based on statistical patterns of abbreviations extracted from 0.7 million projects [6], and (2) unigram frequencies extracted from NLR and SCR for expanding all types of abbreviations (single words and phrases). In comparison, in our paper, we focus on expanding abbreviated phrases using bigram language model definition.

In addition to expansion of abbreviations in the area of software engineering, Zhang et al. [18] used Support Vector Machine (SVM)[19] to expand abbreviations for text-to-speech synthesis. They collected potential abbreviation/full-word pairs by looking for terms that could be abbreviations of full words that occur in the same context. For instance, the pairs in ***svc/service*** *center, heating* ***clng/cooling*** *system, and dry* ***clng/cleaning*** *system* provide evidence that **svc** is an abbreviation of **service**. This SVM-based approach, as a supervised learning technique, requires manual labeling of the abbreviation/full word pairs. Abbreviation expansion also has been studied in the field of health science. Liu et al. [20] exploited task-oriented resources to learn word embedding for clinical abbreviation expansion. Word embedding is a feature learning technique in natural language processing where words from the vocabulary are mapped to vectors of real numbers [21]. For a given abbreviation and its context, the word-embedding approach uses Wikipedia and publications in health science as the training data and the word-embedding method [22] to guess the best abbreviation. Like other approaches mentioned before, the above two approaches focus on retrieving unigrams rather than phrases.

6 CONCLUSIONS

Through the effective expansion of abbreviations, the bigram-based approach can help software maintainers better understand the intent of abbreviations named by original programmers and thus decrease the complicity of software. We have presented the bigram-based approach for automatically expanding an abbreviation to a phrase with multiple unigrams. The experiment results show that the approach has correctly retrieved 78% of the 100 randomly selected phrase abbreviations from eight open source projects. Essentially, our bigram-based model is a knowledge-based model which means we do not rely on predefined abbreviation naming patterns as in existing approaches, therefore, our approach is able to correctly expand the first-time-seen abbreviations invented by programmers. Concerning the future work, we plan to investigate (a) whether or not machine learning techniques can retrieve abbreviated phrases in a more accurate and efficient way, and (b) how natural language processing principles and techniques can be adapted and extended for the summarization of classes and methods in object-oriented programs. The implementation of the bigram-based approach including the 100 phrase abbreviations as well as a web-application of our approach are publicly available[1]

7 ACKNOWLEDGMENT

The work is supported in part by the National Science Foundation #1714261.

REFERENCES

[1] Nioosha Madani, Latifa Guerrouj, Massimiliano Di Penta, Yann-Gael Gueheneuc, and Giuliano Antoniol. 2010. Recognizing words from source code identifiers using speech recognition techniques. In *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*. IEEE. Madrid, Spain, 68–77.

[2] Dawn Lawrie, Dave Binkley, and Christopher Morrell. 2010. Normalizing source code vocabulary. In *Reverse Engineering (WCRE), 2010 17th Working Conference on*. IEEE. Beverly, MA, USA, 3–12.

[3] 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 *Proceedings of the 2008 international working conference on Mining software repositories*. ACM. Leipzig, Germany, 79–88.

[4] Anna Corazza, Sergio Di Martino, and Valerio Maggio. 2012. Linsen: an efficient approach to split identifiers and expand abbreviations. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE. Trento, Italy, 233–242.

[5] Abdulrahman Alatawi, Weifeng Xu, and Dianxiang Xu. 2017. Bayesian unigram-based inference for expanding abbreviations in source code. In *Tools with Artificial Intelligence (ICTAI17), 2017 IEEE 29th International Conference on*. IEEE. Boston, MA, USA.

[6] Weifeng Xu, Dianxiang Xu, Omar El Ariss, Yunkai Liu, and Abdulrahman Alatawi. 2017. Statistical unigram analysis for source code repository. In *Multimedia Big Data (BigMM), 2017 IEEE Third International Conference on*. IEEE. Laguna Hills, CA, 1–8.

[7] Ricardo A Baeza-Yates and Chris H Perleberg. 1992. Fast and practical approximate string matching. In *Annual Symposium on Combinatorial Pattern Matching*. Springer. Tucson, AZ, USA, 185–192.

[8] Abdulrahman Alatawi, Weifeng Xu, and Jie Yan. 2018. The expansion of source code abbreviations using a language model. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*. Volume 2. IEEE, 370–375.

[9] Ashwini I Gadag and BM Sagar. 2016. N-gram based paraphrase generator from large text document. In *Computation System and Information Technology for Sustainable Solutions (CSITSS), International Conference on*. IEEE. Bangalore, India, 91–94.

[10] Daniel Valcarce, Javier Parapar, and Álvaro Barreiro. 2016. Additive smoothing for relevance-based language modelling of recommender systems. In *Proceedings of the 4th Spanish Conference on Information Retrieval*. ACM, 9.

[11] Stanley F Chen and Joshua Goodman. 1999. An empirical study of smoothing techniques for language modeling. *Computer Speech & Language*, 13, 4, 359–394.

[12] Github. 2016. Github.com. (2016). http://www.github.com/..

[13] LD Consortium. 2016. Linguistics data consortium, 2006. google.com. (2016). https://catalog.ldc.upenn.edu/LDC2006T13/.

[14] Sarah Zhang. October 12, 2015. The pitfalls of using google ngram to study language. (October 12, 2015). https://www.wired.com/2015/10/pitfalls-of-studying-language-with-google-ngram/.

[15] Dawn Lawrie, Henry Field, and David Binkley. 2007. Extracting meaning from abbreviated identifiers. In *Source Code Analysis and Manipulation, 2007. SCAM 2007. Seventh IEEE International Working Conference on*. IEEE. Paris, France, 213–222.

[16] Fangfang Feng and W Bruce Croft. 2001. Probabilistic techniques for phrase extraction. *Information processing & management*, 37, 2, 199–220.

[17] Robert Krovetz. 1993. Viewing morphology as an inference process. In *Proceedings of the 16th annual international ACM SIGIR conference on Research and development in information retrieval*. ACM. Pittsburgh, PA, USA, 191–202.

[18] Wei Zhang, Yan Chuan Sim, Jian Su, and Chew Lim Tan. 2011. Entity linking with effective acronym expansion, instance selection, and topic modeling. In *IJCAI*. Volume 2011, 1909–1914.

[19] Corinna Cortes and Vladimir Vapnik. 1995. Support-vector networks. *Machine learning*, 20, 3, 273–297.

[20] Yue Liu, Tao Ge, Kusum S Mathews, Heng Ji, and Deborah L McGuinness. 2018. Exploiting task-oriented resources to learn word embeddings for clinical abbreviation expansion. *arXiv preprint arXiv:1804.04225*.

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

[22] Cataldo Musto, Giovanni Semeraro, Marco de Gemmis, and Pasquale Lops. 2016. Learning word embeddings from wikipedia for content-based recommender systems. In *European Conference on Information Retrieval*. Springer, 729–734.

[1] http://seuresearch.pythonanywhere.com/

Table 8: Phrases Retrieved by Different Methods

#	Abbreviation	Manually Retrieved	UP-NLR	UP-SCR	BP-NLR	BP-SCR
01	ltr	left|to|right	left|to|right	left|to|right	left|to|right	**left|right**
02	fc	fixed|cost	fixed|cost	fixed|cost	fixed|cost	fixed|cost
03	wc	working|calendar	**work|calendar**	**work|calendar**	working|calendar	**work|calendar**
04	dm	display|metrics	**display|may**	display|metrics	**display|mode**	display|metrics
05	cp	class|path	class|path	class|path	class|path	class|path
06	cpe	class|path|entry	**create|project|entry**	**create|project|entry**	**create|project|entry**	**create|project**
07	jme	java|model|exception	java|model|exception	java|model|exception	**java|model**	**java|model**
08	em	entity|mode	entity|mode	entity|mode	**entity|metamodel**	entity|mode
09	lce	loading|collection|entry	**load|collection|entry**	**load|collection|entry**	**loading|contexts|entry**	**loading|collection**
10	ce	context|entry	**can|entry**	context|entry	context|entry	context|entry
11	ps	prepared|statement	prepared|statement	prepared|statement	prepared|statement	prepared|statement
12	ck	cache|key	cache|key	cache|key	cache|key	cache|key
13	uk	unique|key	unique|key	unique|key	unique|key	unique|key
14	fk	foreign|key	foreign|key	foreign|key	foreign|key	foreign|key
15	uk	unique|key	unique|key	unique|key	unique|key	unique|key
16	ad	annotation|descriptor	**add**	annotation|descriptor	annotation|descriptor	annotation|descriptor
17	uc	unique|constraint	unique|constraint	unique|constraint	unique|constraint	unique|constraint
18	pkit	primary|key|iterator	primary|key|iterator	primary|key|iterator	primary|key|iterator	primary|key|iterator
19	annd	annotation|descriptor	**annotation|name|descriptor**	**annotation|name|descriptor**	annotation|descriptor	annotation|descriptor
20	cnfe	class|name|found|exception	class|name|found|exception	class|name|found|exception	class|name|found|exception	class|name|found|exception
21	js	joined|subclass	**joined|simple**	joined|subclass	joined|subclass	joined|subclass
22	fkc	foreign|key|class	foreign|key|class	foreign|key|class	foreign|key|class	foreign|key|class
23	pc	primary|column	**process|class**	**process|class**	**process|class**	**process|class**
24	it	inheritance|table	inheritance|table	inheritance|table	inheritance|table	inheritance|table
25	jc	join|column	join|column	join|column	join|column	join|column
26	sp	second|pass	second|pass	second|pass	second|pass	second|pass
27	fk	foreign|key	**final|key**	**final|key**	foreign|key	foreign|key
28	ee	execution|exception	execution|exception	execution|exception	execution|exception	execution|exception
29	es	entry|set	entry|set	entry|set	entry|set	entry|set
30	gc	graphics|configuration	**graphics|create**	graphics|configuration	graphics|configuration	graphics|configuration
31	st	scale|transform	scale|transform	scale|transform	scale|transform	scale|transform
32	zp	zoom|point	zoom|point	zoom|point	zoom|point	zoom|point
33	pf	page|format	page|format	page|format	**printer|format**	page|format
34	ret	rectangle|type	**range|type**	**range|type**	**range|entities|type**	**range|type**
35	st	scale|transform	scale|transform	scale|transform	**scaley|transform**	scale|transform
36	tt	title|text	title|text	title|text	**title|to**	**text|title**
37	lt	legend|title	legend|title	legend|title	legend|title	legend|title
38	psl	paint|scale|legend	paint|scale|legend	paint|scale|legend	**paint|setbackground|legend**	**paint|scale**
39	ct	composite|title	composite|title	composite|title	composite|title	composite|title
40	bc	block|container	block|container	block|container	block|container	block|container
41	lb	label|block	label|block	label|block	label|block	label|block
42	cp	category|plot	category|plot	category|plot	category|plot	category|plot
43	cp	combined|plot	combined|plot	combined|plot	combined|plot	combined|plot
44	sca	subcategory|axis	subcategory|axis	subcategory|axis	subcategory|axis	subcategory|axis
45	br	bar|renderer	bar|renderer	bar|renderer	bar|renderer	bar|renderer
46	sbr	statistical|bar|renderer	statistical|bar|renderer	statistical|bar|renderer	statistical|bar|renderer	statistical|bar|renderer
47	mmcr	max|min|category|renderer	max|min|category|renderer	max|min|category|renderer	max|min|category|renderer	max|min|category|renderer
48	br	bar|renderer	bar|renderer	bar|renderer	bar|renderer	bar|renderer
49	ta	text|annotation	text|annotation	text|annotation	**to|apply**	text|annotation
50	lg	line|group	line|group	line|group	line|group	line|group
51	ti	table|info	table|info	table|info	table|info	table|info
52	sbr	string|buffer|row	**string|buffer**	**string|buffer**	**string|buffer**	**size|buffer**
53	bd	binary|data	binary|data	binary|data	binary|data	binary|data
54	rs	result|set	result|set	**result**	result|set	result|set
55	isol	isolation|level	isolation|level	isolation|level	isolation|level	isolation|level
56	dbv	database|version	database|version	database|version	database|version	database|version
57	vmn	version|major|number	version|major|number	version|major|number	version|major|number	version|major|number
58	sb	string|buffer	string|buffer	string|buffer	string|buffer	**select|buffer**
59	is	instance|string	instance|string	instance|string	instance|string	instance|string
60	si	select|inquiry	select|inquiry	**string**	select|inquiry	**string|inquiry**
61	sp	system|properties	**system|property**	**select|properties**	system|properties	system|properties
62	ss	schema|system	**system|select**	**schema|select**	schema|system	**system|select**
63	pd	protection|domain	**project|domain**	**project|domain**	**project|domain**	protection|domain
64	ue	unknown|element	unknown|element	unknown|element	unknown|element	unknown|element
65	sysp	system|property	system|property	system|property	system|property	system|property
66	jt	junit|test	junit|test	junit|test	junit|test	junit|test
67	fe	formatter|element	**final|element**	**final|element**	formatter|element	formatter|element
68	fa	formatter|array	**final|array**	**final|array**	**final|array**	formatter|array
69	cm	case|message	case|message	case|message	case|message	case|message
70	as	attribute|string	**attribute|set**	attribute|string	**attribute|setter**	attribute|string
71	ch	class|helper	class|helper	class|helper	class|helper	class|helper
72	ih	introspection|helper	introspection|helper	introspection|helper	introspection|helper	introspection|helper
73	as	attribute|setter	attribute|setter	attribute|setter	attribute|setter	attribute|setter
74	iae	illegal|access|exception	illegal|access|exception	illegal|access|exception	**illegal|exception**	**illegal|access**
75	ite	invocation|target|exception	invocation|target|exception	invocation|target|exception	**invocation|exception**	**invocation|target**
76	nc	nested|creator	nested|creator	**name|creator**	nested|creator	nested|creator
77	ne	nested|element	nested|element	**name|element**	nested|element	nested|element
78	ue	unknown|element	unknown|element	unknown|element	unknown|element	unknown|element
79	nt	name|type	**name|types**	**name|types**	**nested|types**	name|type
80	at	attribute|types	attribute|types	**attribute**	attribute|types	attribute|types
81	ea	enumerated|attribute	enumerated|attribute	**exception|attribute**	enumerated|attribute	enumerated|attribute
82	ie	invoke|exception	invoke|exception	invoke|exception	**instantiationexception|exception**	invoke|exception
83	dow	day|of|week	day|of|week	day|of|week	day|of|week	day|of|week
84	iltr	is|left|to|right	**instance|left|to|right**	**instance|left|to|right**	**instance|left|to|right**	**instance|long|to|right**
85	sfdd	set|first|displayed|date	**set|first|date|displayed**	**set|first|date|displayed**	**set|first|date|displayed**	set|first|displayed|date
86	dfi	date|format|instance	date|format|instance	date|format|instance	date|format|instance	date|format|instance
87	mem	motion|event|mouse	**mouse|mask**	**mouse|mask**	**mouse|event|mask**	motion|event|mouse
88	sga	sequence|generator|annotations	**sequence|generator|annotation**	**sequence|generator|attribute**	**sequence|generator**	**sequence|generator**
89	uisp	user|interface|setting|parameters	**user|interface|set|parameters**	**user|interface|set|parameters**	user|interface|setting|parameters	**user|interface|set|parameters**
90	iltr	is|left|to|right	**indicating|left|to|right**	**indicating|left|to|right**	**indicating|left|to|right**	**indicating|left|to|right**
91	coel	component|orientation|elements|list	**component|elements|list**	**component|elements|list**	**component|elements**	**component|list**
92	dow	day|of|week	day|of|week	day|of|week	day|of|week	day|of|week
93	fdow	first|day|of|week	first|day|of|week	first|day|of|week	first|day|of|week	first|day|of|week
94	ncc	number|calendar|columns	number|calendar|columns	number|calendar|columns	**number|columns|calendar**	number|calendar|columns
95	numcalr	number|calendar|row	number|calendar|row	number|calendar|row	number|calendar|row	number|calendar|row
96	tinum	task|id|number	**to|id|number**	**to|id|number**	**title|id|number**	task|id|number
97	csss	config|smallest|screen|size	**config|screen|size|smallest**	**config|screen|size|smallest**	config|smallest|screen|size	config|smallest|screen|size
98	adidcl	annotation|descriptor|id|class	**add|descriptor|defaults|class|list**	**annotation|descriptor|defaults|class|list**	**annotation|descriptor|defaults|class|list**	**annotation|descriptor|defaults|class|list**
99	ada	annotation|descriptor|access	**access|descriptor|annotation**	annotation|descriptor|access	**access|descriptor|annotation**	annotation|descriptor|access
100	drso	dataset|result|set|order	**dataset|result|specified|order**	**dataset|result|order**	**dataset|result|set|of**	**dataset|rendering|specified|order**

Note: Incorrect expansions are marked with bold font.