AMAP: Automatically Mining Abbreviation Expansions in Programs to Enhance Software Maintenance Tools *

Emily Hill, Zachary P. Fry, Haley Boyd, Giriprasad Sridhara,
Yana Novikova, Lori Pollock and K. Vijay-Shanker
Department of Computer and Information Sciences
University of Delaware
Newark, DE 19716 USA
{hill, fry, boyd, gsridhar, novikova, pollock, vijay}@cis.udel.edu

ABSTRACT

When writing software, developers often employ abbreviations in identifier names. In fact, some abbreviations may never occur with the expanded word, or occur more often in the code. However, most existing program comprehension and search tools do little to address the problem of abbreviations, and therefore may miss meaningful pieces of code or relationships between software artifacts. In this paper, we present an automated approach to mining abbreviation expansions from source code to enhance software maintenance tools that utilize natural language information. Our scoped approach uses contextual information at the method, program, and general software level to automatically select the most appropriate expansion for a given abbreviation. We evaluated our approach on a set of 250 potential abbreviations and found that our scoped approach provides a 57% improvement in accuracy over the current state of the art.

Categories and Subject Descriptors: D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement; D.2.7 [Software Engineering]: Coding Tools and Techniques

General Terms: Human Factors, Reliability

Keywords: Automatic abbreviation expansion, software maintenance, program comprehension, software tools

1. INTRODUCTION

When writing software, developers often use abbreviations in identifier names, especially for identifiers that must be typed often and for domain-specific words used in comments. In some cases, the abbreviated form of a word is so prevalent that it occurs more often than the expanded form. For example, the word 'number' occurs only 4,314 times in the Java 2 Platform, while its abbreviation 'num' occurs 5,226 times.

*This material is based upon work supported by the National Science Foundation under a Graduate Research Fellowship and Grant No. CCF-0702401.

MSR'08, May 10-11, 2008, Leipzig, Germany.

Unfortunately, most existing software tools that use the natural language information in comments and identifiers do nothing to address abbreviations, and therefore may miss meaningful pieces of code or relationships between software artifacts. For example, if a developer is searching for string handling code, she might enter the query 'string'. If the abbreviation 'str' is used in the code instead of 'string', the search tool will miss relevant code.

Thus, techniques for expanding abbreviations can improve the effectiveness of a variety of language-based software tools such as concern location [6, 13, 17, 19], documentation to source code traceability [1, 12], or other software artifact analyses [2, 16]. Automatically expanding abbreviations will give these tools access to words and associated meanings that were previously meaningless sequences of characters.

In this paper, we define a *token* to be a sequence of alphabetic characters delimited by any non-alphabetic token such as spaces or underscores. We refer to any token that is not found in an English dictionary as a *non-dictionary word*. We use the term *short form* to refer to an abbreviation, and *long form* for its corresponding full word expansion.

One simple way to expand short forms in code is to manually create a dictionary of common short forms [16]. Although most developers understand that 'str' is a short form for 'string', not all abbreviations are as easy to resolve. Consider the abbreviation 'comp'. Depending on the context in which the word appears, 'comp' could mean either 'compare' or 'component'. Thus, a simple dictionary of common short forms will not suffice. In addition, manually created dictionaries are limited to abbreviations known to the dictionary builders.

The hypothesis driving our work is that automatically mining short forms from the program itself can identify the most appropriate expansions of short forms within the context of the individual occurrences. In this paper, we present an automatic technique to mine short and long forms from a large set of programs, such that abbreviations in a program can be automatically expanded to the most appropriate long form in the context of their occurrences. We evaluate the effectiveness of our automatic program abbreviation expansion technique by comparing it with Lawrie, Feild, and Binkley's technique [9] and two variations of a most frequent expansion-based approach. Our scoped approach provides a 57% improvement in accuracy over the current state of the art. We are able to find appropriate expansions for single-word abbreviations including acronyms, prefixes, and dropped letter short forms, as well as multi-

word abbreviations including acronyms. Although our current work focuses on Java programs predominantly written in English, our approach can be applied to any programming and natural language combination.

The major contributions of this paper are:

- A detailed analysis of the types of abbreviations found in software and the challenges in automatically expanding them
- A fast and effective technique for automatically expanding program abbreviations of many types
- An experimental evaluation comparing the accuracy of four techniques: the only known existing technique [9], our scoped approach, and two derivative approaches

2. TYPES OF NON-DICTIONARY WORDS IN CODE

There are many types of non-dictionary words used in program identifiers, and these non-dictionary words are not limited to abbreviations. We have found abbreviations generally fall into two categories: single-word and multi-word.

2.1 Single-Word Abbreviations

Single-word abbreviations are short forms whose long form consists of a single word. This is in contrast to non-dictionary words such as acronyms, whose long forms expand into multiple words. We have identified two major types of single word abbreviations in programs: prefixes and dropped letters. *Prefix* short forms are formed by dropping the latter part of a long form, retaining only the few beginning letters. Examples of prefixes include 'attr' (attribute), 'obj' (object), and 'param' (parameter). A subset of prefix short forms are single letter prefixes. *Single letter prefixes* are predominantly used for local variables with very little scope outside a class or method [10]. Examples include 'i' (integer) and 'e' (exception).

The second type of single-word abbreviation is dropped letter. *Dropped letter* short forms can have any letters but the first letter removed from the long form. Examples include 'evt' (event), 'msg' (message), and 'src' (source). Dropped letter short forms are actually a super set of prefix short forms, although they can easily expand to a much larger set of long forms. For example, the abbreviation 'org' can expand to be a prefix of 'organization' or be the less probable dropped letter 'original'. Automatic abbreviation expansion techniques must therefore be selective in expanding dropped letter abbreviations to potential long forms.

2.2 Multi-Word Abbreviations

Multi-word abbreviations are short forms that when expanded into long form consist of more than one word. The most common are acronyms, which arguably belong in a class of short forms separate from abbreviations. For the purposes of this work, we consider acronyms to be a special type of multi-word abbreviation. *Acronyms* consist of the first letters of the words in the long form. Acronyms can be so widely used that the long form is rarely seen, such as 'ftp', 'xml', or 'gif'. Some uses of acronyms are very localized, such as what we call type acronyms. When creating local variables or naming method parameters, a common naming scheme is to use the type's abbreviation. For example, a variable of the type ArrayIndexOutOfBoundsException may be abbreviated 'aiobe', or StringBuffer as 'sb'.

The second type of multi-word abbreviation includes more than just the first letters of the long form. A *combination multi-word* may combine single-word abbreviations, acronyms, or dictionary words. Examples include 'oid' (object identifier), 'println' (print line), and 'doctype' (document type). By definition, combination multi-words must contain more than two letters, otherwise the short form would be an acronym. As with the relationship between prefixes and dropped letters, acronyms are a subset of combination multi-words.

2.3 Other Types of Short Forms

Aside from abbreviated words, one of the most common forms of non-dictionary words in code are multiple words with no clearly delineated word boundaries. Most identifiers that consist of multiple words contain word boundaries by varying upper and lower case letters (i.e., camel casing) or by using non-alphabetic characters as in the examples ASTVisitor, stringBuffer, and TARGET_WINDOW. However, a programmer may not delineate word boundaries because: (1) the boundaries are trivial for a human to recognize, such as in 'keystore' or 'threadgroup'; (2) the programmer favored typing fewer letters over general readability; or (3) the words appear so often together that the programmer may not realize the compound word does not exist in English. In the last case, the joined words may be considered one word to most programmers, but English dictionaries currently include only the separate parts. We have found that lack of word boundaries is especially common for *collocations*—words that often occur adjacent to one another and represent a conventional way of saying things [11]. Examples include 'filesize', 'saveas', and 'dataset'.[1]

Misspellings are also present in identifiers, although less so than for comments. Examples include 'instanciation' (instantiation), and a strike 'trought' (through) font format. Other types of non-dictionary words include mathematical notation, such as for vector indices or notation specific to scientific equations, and Hungarian notation [18]. More common for C-based languages than Java, Hungarian notation suggests appending the first letter of the data type to every variable name. Finally, some identifiers are just improbably named: 'zzzcatzzzdogzzz'. When variable names are selected with little relevance to the underlying code, deriving meaning can be impossible both for humans and automated mining techniques.

3. AUTOMATIC ABBREVIATION EXPANSION

Automatically expanding abbreviations requires the following steps: (1) identifying whether a token is a non-dictionary word, and therefore a short form candidate; (2) searching for potential long forms for the given short form; and (3) selecting the most appropriate long form from among the set of mined potential long form candidates.

For some applications, a completely automated approach may be unnecessary and the final step of selecting the most appropriate long form can be left to the human user. Examples include a program comprehension tool that automatically presents the developer with potential long forms when given an unfamiliar abbreviation as input; or a query expansion mechanism that uses human feedback to determine appropriate query expansions, which may include short forms

[1]Components of these types of multi-words are called soft words by Lawrie, Feild, and Binkley [9].

for a long form given in the query. However, for general search or automated maintenance tools, a fully automatic approach may be more appropriate. For the remainder of this paper we focus on completely automatic abbreviation expansion.

3.1 Observations and Challenges

To develop our automatic abbreviation expansion technique, we analyzed short forms and their corresponding long forms in 15 open source Java programs. Based on our manual inspection, we made the following observations that must be taken into account when automatically mining abbreviation expansions:

Good dictionaries are hard to find. The most prevalent available English dictionaries are used for spell checking, and may include proper nouns, common abbreviations, and contractions, all of which may occur in software. However, some dictionary words are less likely to occur in code. For example, observing the token 'io' in software is much more likely to stand for 'input output' than the proper noun 'Io'. In addition, legitimate English words may be used for abbreviations in code, such as 'char', 'tab', or 'id'. Thus, there is a trade off in including too many or too few words in a dictionary. Too few words causes the automatic expansion to attempt to find long forms for legitimate words; too many words causes even legitimate short forms to be classified as dictionary words.

Short form type is impossible to determine a priori. If it were possible to automatically identify the short form type, it would be easier to narrow down the list of potential long forms, and therefore more accurately select the appropriate long form. Unfortunately, abbreviations are short forms for longer words, and are by their very nature less unique than the long forms themselves. Thus, the same sequence of three characters may represent different long forms depending on the context. For example, 'def' can refer to 'definition', 'default', or even 'defect'. In one instance, we even observed that the acronym 'dc' was used to represent both 'dynamic color' and 'duration color' in different branches of the same method. Thus, we cannot rely solely on the abbreviation type to eliminate unrealistic long form candidates.

The shorter the short form, the more potential long form candidates. This observation presents one of the more frustrating aspects of the automatic abbreviation expansion problem. By definition, the shorter the short form, the more potential long forms it could match. For example, a single letter abbreviation 'i' could conceivably match any dictionary word beginning with the letter 'i'; whereas 'int' is likely to match 'integer', 'interface', or 'interrupt'; and 'interf' will match 'interface'. However, most abbreviations are short, consisting of just 1–3 letters. Thus, the majority of short forms represent the most difficult instances of the automatic abbreviation expansion problem.

Some abbreviation types have more long form candidates. Specifically, acronyms and prefixes have fewer long form candidates than dropped letters and combination multi-words. For example, the prefix 'str' is likely to be 'string' or 'stream', whereas the dropped letter 'str' could match long forms 'substring', 'store', 'september', or 'saturn'. Thus, automatic expansion techniques should take long form accuracy of abbreviation type into account when choosing between potential long forms.

3.2 State of the Art

To our knowledge, Lawrie, Feild, and Binkley [9] are the only other researchers to present and evaluate techniques to address the problem of automatically expanding abbreviations that occur in program identifiers. In their earlier paper [4], Feild, Lawrie and Binkley evaluated three automated techniques for splitting identifiers that are not easily split by camel-casing or underscore clues left by the programmer. By first splitting the identifiers into their constituent "words", their abbreviation analysis can focus on the individual "words" comprising each identifier.

More recently, Lawrie, Feild, and Binkley (LFB) [9] presented a strategy for automatically expanding abbreviations used in identifiers by first extracting lists of potential expansions as words and phrases, and then performing a two-stage expansion for each abbreviation occurrence in the code. They create several different lists to be used during expansion of an identifier occurrence. For each function f in the program, they create a list of words contained in the comments before or within the function f or in identifiers with word boundaries (e.g., camel casing) occurring in f, and a phrase dictionary created by running the comments and multi-word-identifiers through a phrase finder [5].

In addition to the lists for each function, they create a list of programming language specific words as a stop word list. Stemming and the stop word list are used during extraction to improve accuracy. The first letter of each phrase is used to build acronyms. Expansion of a given non-dictionary word occurrence in a function f involves first looking in f's word list and phrase dictionary, and then in a natural language dictionary. A word is said to be a potential expansion of an abbreviation when the abbreviation starts with the same letter and every letter of the abbreviation occurs in the word in order.

The LFB [9] technique returns a potential expansion only if there is a single possible expansion. They leave the problem of choosing among multiple potential expansions found at either stage as future work. When they manually checked a random sample of 64 identifiers requiring expansion (from a set of C, C++, and Java codes), one third were correctly split and expanded. Of the identifiers correctly split, 58% of the one–two letter forms were expanded correctly and 64% of the over-two letter forms. Thus, only approximately 20% (60% of 33%) of the identifiers were expanded correctly. In their other quantitative study of all identifiers in their 158-program suite of over 8 million unique terms, only 7% of the total number of identifier terms were expanded by their technique; these expansions were not checked for correctness. These low precision results motivate a closer look at alternative strategies for expansion. In addition, sets of potential expansions for a given occurrence in their study ranged from 1 to 6735, demonstrating the need for a heuristic for choosing the most appropriate expansion for a given occurrence.

Somewhat related work includes the work on restructuring program identifier names to conform to a standard in both the lexicon of the composed terms and the syntactic form of the overall identifier composition of terms [3]. Identifiers are split, and then a match between a standard dictionary and synonym dictionary and the identifier components is attempted. When no match is found, the user is prompted for help. No automatic abbreviation expansion is attempted.

There exist acronym expansion techniques created for use

in written English text [8, 14]; however, their premise does not hold for software due to their reliance on textual patterns that do not occur in code and do not apply in the context of the syntactic structure of a program.

4. THE SCOPED APPROACH

As outlined in the beginning of Section 3, the LFB approach only addresses the first two steps of identifying non-dictionary words and potential expansions [9]. When faced with the final step of choosing between equally likely long forms, their current approach returns nothing. With our scoped approach, we attempt to effectively solve all three steps of the automatic abbreviation expansion problem.

Also in contrast to LFB, we never attempt to match short forms to an English dictionary of words, only those dictionary words appearing within the scope of the software. This is a direct consequence of our observation that word lists for computer science are nonexistent, and many English dictionaries include too many words. Also, the only hand-tuned word lists used in our approach are a stop word list and list of common contractions. Our list of common abbreviations is automatically derived from software, but could be improved with a hand-tuned common abbreviation list, especially for production systems.

Our automatic long form mining technique is inspired by the static scoping of variables represented by a compiler's symbol table. When looking for potential long forms, we start at the closest scope to the short form, such as type names and statements, and gradually broaden our scope to include the method, its comments, and the class comments. If our technique is still unsuccessful in finding a long form, we attempt to find the most likely long form found within the program and in Java SE 1.5. With each successive scope we include more general, i.e., less domain specific, information in our long form search.

In our current work, we assume a short form has the same long form for an entire method. Although infrequent, it is possible for a short form to have multiple long forms within a method. To handle such cases, our approach could be extended to assume a short form has the same long form for only block or even statement level scope.

4.1 Method-level Matching

The core of our approach is our long form search technique within a method. In this section, we first describe how we search for each type of long form within a method, define how we select the long form from many potential long forms, and how we attempt to expand short forms initially missed at the method scope.

4.1.1 Single-Words

The first step in searching for long forms is to construct a regular expression *pattern* from the short form and then use the pattern to search for long form candidates over different parts of the method body text. Our single-word search approach is presented in Algorithm 1, and the patterns used for each type of short form are described below.

Stepping through Algorithm 1, line 6 prevents (1) searching for unlikely dropped letter long forms and (2) expanding short forms with many consecutive vowels as a single-word. The first three predicates in line 6 restrict the search for dropped letter long forms to only those short forms that are longer than 3 letters or composed of all consonant letters with an optional leading vowel. We restrict the dropped letter pattern search because the pattern can greedily match many incorrect expansions. For example, if left unchecked, dropped letter may incorrectly expand 'lang' to 'loading', 'br' to 'bar', or 'mtc' to 'matching'. The last predicate of line 6 ensures that we do not try to expand short forms with many consecutive vowels as a single-word. Most short forms consisting of consecutive vowels expand into multi-word long forms; consider 'gui' (graphical user interface), 'ioe' (invalid object exception), or 'poa' (portable object adaptor).

Algorithm 1 Searching for single-word long forms. Quotes are used to indicate regular expressions.

```
1: Input: potential short form, sf
2: Input: regular expression to match long form, pattern
3: Input: method body text, method comments
4: Input: class comments (Prefix only)
5: Output: long form candidates, or null if none
6: if ((prefix pattern) or (sf matches "[a-z][^aeiou]+") or
   (length(sf) > 3)) and
   (sf does not match "[a–z][aeiou][aeiou]+") then
7:    In the following, when a unique long form is found, return.
8:    Search JavaDoc comments for "@param sf pattern"
9:    Search TypeNames and corresponding declared variable names for "pattern sf"
10:   Search MethodName for "pattern"
11:   Search Statements for "pattern sf" and "sf pattern"
12:   if length(sf) ≠ 2 then
13:      Search method words for "pattern"
14:      Search method comment words for "pattern"
15:   end if
16:   if (length(sf) > 1) and (prefix pattern) then
17:      Search class comment words for "pattern"
18:   end if
19: end if
```

Lines 7–19 of the algorithm describe the search process. If at any line a unique long form is found, the algorithm immediately returns. In line 8, we first search for the short form and the pattern in the method's Java Doc comment. If unsuccessful, in line 9 we look for the short form and the pattern appearing together in a variable declaration and its type. Next we search the method name for the pattern in line 10. In line 11 we continue searching for the pattern and the short form appearing within the same statement.

In line 12 we restrict our search of the general method text and comments to short forms of 3 letters long or more because short forms that are two letters long (1) are most likely to be multi-words and (2) are capable of matching many different words. Since we do not search beyond method scope for single letter prefixes, we also search the method text and comments for single letter prefixes. Thus, in lines 13–14 we search for the pattern in the method words and method comment words if the short form is not of length two.

Lastly, if the pattern is a prefix and the short form is longer than a single letter, we search the class comments for the pattern in line 17. Since single letter prefix short forms are unlikely to have scope beyond a method, and since the single letter prefix pattern may match so many long forms, we do not attempt to match single letter prefix patterns to the class comments. Likewise, since the dropped letter pattern is so greedy, we do not search for dropped letter long forms in the class comments.

Algorithm 2 Searching for multi-word long forms. Quotes are used to indicate regular expressions.

```
1: Input: potential short form, sf
2: Input: regular expression to match long form, pattern
3: Input: method body text, method comments
4: Input: class comments (Acronym only)
5: Output: long form candidates, or null if none
6: if (acronym pattern) or (length(sf) > 3) then
7:    In the following, when a unique long form is found, return.
8:    Search JavaDoc comments for "@param sf pattern"
9:    Search TypeNames and corresponding declared variable names for "pattern sf"
10:   Search MethodName for "pattern"
11:   Search all identifiers in the method for "pattern" (including type names)
12:   Search string literals for "pattern"
      {At this point we have searched all the possible phrases in the method body}
13:   Search method comment words for "pattern"
14:   if acronym pattern then
15:      Search class comment words for "pattern"
16:   end if
17: end if
```

Prefix Pattern.

The first step in searching for prefix long forms is to construct a regular expression from the short form. The prefix pattern is thus the short form followed by the regular expression "[a–z]+": "sf[a–z]+". The letter 'x' is a special case: if a short form begins with 'x', the expression "e?x" is added to the beginning of the pattern. The pattern is then used as input to Algorithm 1 to search for long forms.

Dropped Letter Pattern.

The regular expression pattern for dropped letter is much less conservative than the pattern used for prefixes. The dropped letter pattern is constructed by inserting the expression "[a–z]*" after every letter in the short form. Let $sf = c_0, c_1, ..., c_n$, where n is the length of the short form. Then the dropped letter pattern is c_0[a–z]*c_1[a–z]*...[a–z]*c_n.

4.1.2 Multi-Words

As with single-words, our approach for finding multi-word long forms searches increasingly broader scopes until we find a long form candidate that matches the pattern. However, because multi-word patterns must search over spaces, it is important to limit how far the pattern should extend. For example, with a naive pattern the short form 'il' could match the phrase "**i**t is important to **l**imit" in the previous sentence. Thus, we preprocess the method body text and comments so that we do not search for long forms beyond variable declarations and method identifier boundaries. We also split comments and string literals into phrases using punctuation ([?!,;]). So that abbreviations like 'val' are not expanded to 'verify and load', we remove common stop words from the method body text and comments.

Our multi-word search approach is presented in Algorithm 2. Line 6 ensures that we do not search for many incorrect combination word long forms. Combination word patterns are much less conservative than acronyms, and can frequently match incorrect expansions. Thus, we restrict our search to short forms of length 4 letters or more. This threshold will cause our technique to miss some legitimate 3-letter combination word expansions, such as 'oid' (object identifier), but we feel it is necessary to restrict our search to find only the most likely long forms. It should be noted that 3-letter combination word abbreviations are not very common in practice. In the random sample of 250 non-dictionary words used in our evaluation, only 1 short form fell into this category. Based on this sample, we expect 3-letter combination word abbreviations to account for only 4% of all combination words and just 0.4% of all non-dictionary words.

As with single-words, our technique searches for multi-word long form candidates first in Java Doc, type names, and the method name in lines 8–10. We were unable to search for multi-words in statements due to run time complexities of the regular expression. Next in lines 11–12 we search the method identifiers and string literals for the pattern, followed by method comments in line 13. Because expansions for well understood short forms in the context of the class may not occur within the method text and comments, we also search for acronym long forms in the class comments in line 15. As with dropped letter, we do not search for combination word patterns in the class comments because the pattern can match many incorrect long forms.

Acronym Pattern.

The regular expression pattern used to search for acronym long forms is simply constructed by inserting the expression "[a–z]+[ ]+" after every letter in the short form. Let $sf = c_0, c_1, ..., c_n$, where n is the length of the short form. Then the acronym pattern is c_0[a–z]+[ ]+c_1[a–z]+[ ]+...[a–z]+[ ]+c_n. As with prefixes, the letter 'x' is a special case. When forming the acronym pattern, any occurrence of 'x' in the short form is replaced with the expression "e?x." This enables our technique to find long forms for acronyms such as 'xml' (**ex**tensible markup language).

Combination Word Pattern.

The pattern to search for combination word long forms is constructed by appending the expression "[a–z]*?[ ]*?" to every letter of the short form. Let $sf = c_0, c_1, ..., c_n$, where n is the length of the short form. Then the combination word pattern is c_0[a–z]*?[ ]*?c_1[a–z]*?[ ]*?...[a–z]*?[ ]*?c_n. The pattern is constructed such that only letters occurring in the short form can begin a word. This keeps the pattern from expanding short forms like 'ada' with 'adding machine'. We use a less greedy wild card to favor shorter long forms with fewer spaces, such as 'period defined' for 'pdef', rather than 'period defined first'.

4.1.3 Putting it all together

With a slightly different technique to search for long forms of each abbreviation type, we now have to combine them together to output a single long form. The first step is identifying the order to apply the expansion techniques. Within the single- and multi-word types, it should be obvious that acronyms should be matched before combination words and prefixes before dropped letter, since the greedier patterns will match all the long forms that the more conservative patterns match. However, we were not immediately sure in what order to search for acronym and prefix or dropped letter and combination word. After manually inspecting hundreds of example long forms for 15 open source Java pro-

grams, we concluded that the best order to apply the long form search techniques is: acronym, prefix, dropped letter, and combination word. If none of the abbreviation type expansion techniques match locally within the method, we attempt to match the short form to common contractions, followed by our most frequent expansion (MFE) technique.

4.1.4 Handling multiple matches

Before presenting our MFE technique, we must address how to handle short forms whose pattern matches multiple long form candidates within the same method. Within broader scopes such as method or comments, it is possible for a single abbreviation type pattern to match many potential long forms. For example, the prefix pattern for 'val' may match 'value' as well as 'valid' in a method comment. Our technique for selecting between multiple long forms is as follows:

Step 1. Use the long form that most frequently matches the short form's pattern in this scope. For example, if 'value' matched the prefix pattern for 'val' three times and 'valid' only once, return 'value'.

Step 2. Group words with the same stem [15] and update the frequencies accordingly. For example, if the words 'default' (2 matches), 'defaults' (2 matches), and 'define' (2 matches) all match the prefix pattern for 'def', group 'default' and 'defaults' to be the shortest long form, 'default' (4 matches), and return the long form with the highest frequency.

Step 3. If there is still no clear winner, continue searching for the pattern at broader scope levels. For example, if both 'string buffer' and 'sound byte' match the acronym pattern for 'sb' at the method identifier level, continue to search for the acronym pattern in string literals and comments. We store the frequencies of the tied long forms so that the most frequently occurring long form candidates are favored when searching the broader scope.

Step 4. If all else fails, abandon the search and let MFE select the long form. At this point we stop searching for long form candidates of different abbreviation types. For example, if a prefix pattern has already found long form candidates, we avoid finding dropped letter long form candidates by halting the search for a given short form within a method.

4.2 Most Frequent Expansion (MFE)

Our most frequent expansion (MFE) technique leverages successful local expansions to help derive long forms for short forms that would otherwise be missed. Although not all short form expansions are correct, the assumption is that taken over the entire program, the most frequently occurring long form will be the correct one.

We calculate MFE by running our local abbreviation expansion approach over the entire program. Then, for each short form, we count how many times the short form was matched to a given long form. We calculate the relative frequency that a short form was expanded to each long form. The long form with the highest relative frequency is considered to be the most frequent expansion. As with the final step in selecting between potential long forms, we also group long forms with the same stem when creating the MFE list.

However, occasionally an incorrect long form may be considered the most likely expansion. To avoid this, we only consider long forms that were matched for more than half (0.5) of the short form matches, and for short forms that were matched at least 3 times in the entire program.

Short Form	Long Form	Relative Frequency
int	integer	0.821
impl	implement	0.840
obj	object	1.000
pos	position	0.828
init	initial	0.955
len	length	0.990
attr	attribute	1.000
num	number	0.985
env	environment	0.972
val	value	0.894
str	string	0.881
buf	buffer	0.992
ctx	context	0.962
msg	message	0.977
cs	copyright sun	0.665
var	variable	0.974
elem	element	1.000
param	parameter	0.992
decl	declare	0.920
arg	argument	0.964

Table 1: Top 20 entries in the most frequent expansion (MFE) list for Java 5.

We apply our MFE technique at two levels: the program level and the more general Java level. The program level ideally helps expand domain matches. For example, an open source implementation of Guitar Pro has frequent occurrences of the short form 'gp'. Although 'gp' was incorrectly matched to 'graphics' 8 times, our technique correctly expanded 'gp' to 'guitar pro' with a relative frequency of 0.68.

In addition to program MFE information, we also use more general programming knowledge from the Java API implementation. The top 20 entries of our MFE list for Java 5 are presented in Figure 1. If an unexpanded short form is not present in the program MFE we look for it at the more general Java level. The Java MFE list can be calculated ahead of time, or even run over a larger set of Java programs, rather than just the Java API implementation. If our scoped approach were to be applied in practice, this is the stage where a hand-tuned MFE list could be used to improve accuracy.

It is possible that some frequently occurring short forms may never occur with the correct long form, or that the short form is so prevalent in the domain that the long form does not appear anywhere at all. Examples include common acronyms such as 'xml' or domain-specific terminology such as 'ast' in compilers or 'rsa' in encryption. One solution to this problem is to hand-tune the MFE list for the most frequently occurring short forms, or train the JavaMFE approach on a larger set of Java programs as mentioned above. Another solution would be to mine potential long forms beyond the scope of Java programs by utilizing online documents related to Java or computer science in general. For example, abbreviation expansion techniques created for English [8, 14] could be used to mine potential long forms from online textbooks in computer science.

Program	Version	Developers	NCLOC	Types	Methods	# Non-Dictionary
Liferay Portal	4.3.2	94	393,802	4,050	39,747	188,955
OpenOffice.org Portable	2.2.1	8	372,807	4,213	20,374	274,969
iText.NET	1.4-1	2	361,403	4,465	34,141	217,965
Tiger Envelopes	0.8.9	1	350,046	3,005	19,706	191,787
Azureus	3.0.3.0	7	335,515	5,335	28,255	193,757

Table 2: Programs used in the evaluation.

4.3 Implementation

Our technique is fully automatic and is implemented as a Java Eclipse plugin with command line scripts for the MFE calculations, which could easily be added to the plugin in the future. The current implementation is designed for batch processing, but could be incrementally updated or run in the background to support software maintenance tools.

Due to computational issues involved in Java regular expressions, we limited our non-dictionary words to length 10 or less. This rules out some non boundary words that would otherwise be expanded to combination word long forms, such as 'numericfield', but relieves our implementation from attempting to find long forms for non-dictionary words like 'pppppppppppq' (which we actually came across in an open source project). However, we do not feel that limiting the short form length to 10 impairs our technique, since most abbreviations are considerably less than this limit.

Our approach uses a number of word lists and dictionaries, some of which have been hand-tuned for software. For example, we have removed any words from our stop list that could be content words in software, such as 'face', 'case', and 'turn'. The word lists used in our implementation as well as descriptions of how they were derived are available online.[2]

5. EVALUATION

We evaluated our automatic abbreviation expansion technique with two research questions in mind:

1. How does our technique compare to the program and Java MFE approaches?
2. How does our technique compare to the state of the art LFB [9] approach?

5.1 Experiment Design

5.1.1 Variables and Measures

The independent variable is the abbreviation expansion technique, which we evaluated by measuring the accuracy of each technique in finding the correct long forms for a human-annotated gold set of non-dictionary words.

To evaluate how important local scope and domain information is to our expansion technique, we compared our approach to storing and using just the program (ProgMFE) or Java MFE (JavaMFE) information. To expand a short form using an MFE technique, we run our local expansion algorithm on either the entire program or Java once, and calculate the MFE list. The list is used to expand every short form. Thus, the ProgMFE approach expands every short form for a given program to the same long form, and the JavaMFE approach expands every short form to the same long form independent of program.

[2]`http://www.cis.udel.edu/~hill/amap`

We also compared our expansion technique with existing work by implementing the Lawrie, Feild, and Binkley (LFB) technique based on their description [9]. According to their paper, we implemented LFB to search for dropped letter (including prefix) and acronym expansions in dictionary words appearing in the method or comment where the short form occurs. Then, if the short form is not a Java reserved word, an ispell dictionary of words is searched for expansions. Although the paper mentions using maximum likelihood estimation (MLE)[3] to select between multiple long form candidates in the future, the existing approach has no mechanism to select between multiple long form candidates. Therefore, if there is more than one long form found in the method and comment, or in the dictionary, no long form is returned.

There was one aspect of the LFB approach that we were unable to implement at this time. In contrast to our approach, which searches for no boundary short forms based on our combination word pattern, LFB handles combination long forms by recursively searching for possible places to split the short form. For example, the identifier splitting approach would split the non-dictionary word zeroindegree into zero-in-degree. To split these non-dictionary words, LFB searches for successively shorter prefixes and suffixes of dictionary words and a list of common abbreviations. We did not have access to this list of common abbreviations, and felt it might be unfair to evaluate the effectiveness of their approach with a substituted list. The only short form types this affects are combination word (CW), thus we only compare our technique to LFB for prefix (PR), dropped letter (DL), and acronym (AC) short form types.

The dependent variable in our study is the effectiveness of each technique, measured in terms of accuracy. Accuracy is determined by counting the number of short forms that are correctly expanded from a gold set of non-dictionary words. If the non-dictionary word is a short form, the technique should output the corresponding long form, otherwise the technique should output nothing (no long form). To calculate accuracy, we divide the total number of correctly expanded non-dictionary words by the total number of non-dictionary words in the gold set.

5.1.2 Subjects

The subjects in our study are short forms originating from Java programs. We selected 5 open source Java programs from different domains and with different numbers of developers. We chose our programs with approximately equal lines of code (between 300-400K) to avoid bias during our random selection of non-dictionary words for the gold set. Table 2 shows characteristics of the subject programs.

[3]Our notion of most frequent expansion is closely related to maximum likelihood estimation [11]. However, because we filter on the number of matches and require the long forms in our MFE list to be matched in the majority of cases, MFE is not identical to MLE.

	Abbreviation Type						Total	
	AC	**PR**	**SL**	**DL**	**CW**	**OO**	**NCW**	**Total**
Count	49	59	64	9	23	46	227	250
Percent	19.6%	23.6%	25.6%	3.6%	9.2%	18.4%	90.8%	100%

Table 3: Distribution of short forms in abbreviation types for the gold set. The 5 abbreviation types are acronym (AC), prefix (PR), single letter prefix (SL), dropped letter (DL), combination word (CW), and other (OO). The column 'NCW' totals all abbreviation types but CW.

Liferay Portal. Liferay Portal is an open source portal framework for integrated Web publishing and content management with an enterprise service bus and service-oriented architecture. Because Liferay Portal is a secure portal platform, the program text contains terms from security in addition to web publishing and content management.

OpenOffice.org Portable. The goal of the PortableApps.com project is to make applications portable by taking existing applications and packaging them to run from a portable device (e.g., USB flash drive). The program text contains terms relating to document editing, run time GUI management, and mathematical calculations.

iText.NET. iText is an open source library for creating and manipulating PDF, RTF, and HTML files in Java. For example, iText allows developers to extend the capabilities of their web server applications in order to generate a PDF document. The program text contains terms related to reading and converting PDF files.

Tiger Envelopes. Tiger Envelopes is an open source personal mail proxy that automatically encrypts and decrypts mail. The program text contains terms related to encryption and mail clients.

Azureus. Azureus is a Java-based client for sharing files using the BitTorrent file-sharing protocol. The program text contains terms related to file management, runtime GUI management, and networking.

We randomly selected 250 non-dictionary words from the 5 subject Java programs. Two human annotators who had no knowledge of our mining technique manually inspected each short form candidate to identify the abbreviation type and the most appropriate long form for the given context. This served as our gold set. Some non-dictionary words were not abbreviations at all, such as mathematical variables or the program name, and were marked as abbreviation type 'other' (OO). The distribution of short forms across abbreviation types is listed in Table 3. We consider an occurrence of a non-dictionary word to be unique per method. Thus, for any given method, we assume that all instances of a short form have the same long form.

5.1.3 Methodology

We ran each of our implemented tools including our Scope technique, LFB, JavaMFE, and ProgMFE, on the entire set of 250 non-dictionary words. We compared the output of each tool with the gold set. If the long form in the gold set and the technique's automatically determined long form have the same stem according to Porter's stemmer [15], then the expansion is considered to be correct. If the non-dictionary word was not an abbreviation and the technique output no long form, then the expansion is also considered to be correct. We computed the accuracy for each type of abbreviation for each tool. We then computed the accuracy of each tool for short forms by length from one character to 10 characters long, aggregated over all types of abbreviations.

Type	**LFB**	**JavaMFE**	**ProgMFE**	**Scope**
CW	0.000	0.304	0.000	0.174
DL	0.111	0.778	0.667	0.778
OO	0.826	0.652	0.609	0.478
AC	0.285	0.122	0.408	0.469
PR	0.322	0.728	0.746	0.797
SL	0.297	0.313	0.594	0.688
NCW	**0.401**	0.467	0.599	**0.630**
Total	0.364	**0.452**	**0.544**	**0.588**

Table 4: Percent correct expansions for each technique and abbreviation type. To fairly compare our Scope technique to LFB, use the NCW total accuracy.

5.2 Threats to Validity

We attempted to gather a domain-independent gold set of short forms by selecting large programs with different functionality. However, many of our subject programs involve a security component, and our gold set includes a number of short forms related to network security and encryption. Therefore the results of the study may not generalize to all program domains. In addition, because our technique is developed on Java programs predominantly written in English, the results of the study may not generalize to all programming language and natural language combinations.

As with any subjective task, it is possible that the human annotators did not identify the correct long form for a given short form. In some instances, a single short form may be interpreted as different long forms by different developers. To limit this threat the gold set short forms were mapped by two independent developers who were unfamiliar with any of the techniques used in this study. When the appropriate long form was unclear, the non-dictionary word was classified as type 'other' (OO).

5.3 Results and Analysis

We present the accuracy results for our experiment in Table 4. Overall, our approach provides a 57% improvement in accuracy over the current state of the art, LFB, when non-combination-word (NCW) short forms are considered. In addition, both the JavaMFE and ProgMFE had higher accuracy overall than LFB. Because LFB outputs nothing rather than choose between two potential long forms, a significant portion of the technique's correct results are due to correctly *not* identifying long forms. This is evidenced by LFB's high accuracy, over 80%, for the other (OO) category.

The accuracy results in Table 4 also demonstrate the effectiveness of using scope in correctly identifying long forms. As illustrated in Figure 1, there is a steady increase in accuracy as more local context information is used, from JavaMFE to ProgMFE to our fully contextualized Scope approach.

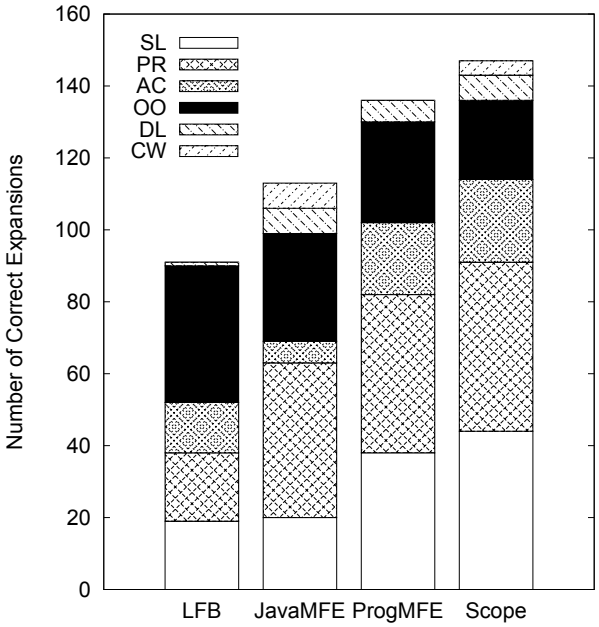

Figure 1: Number of correct expansions by abbreviation type. To fairly compare all the techniques to LFB, do not include the top most bar for combination word (CW).

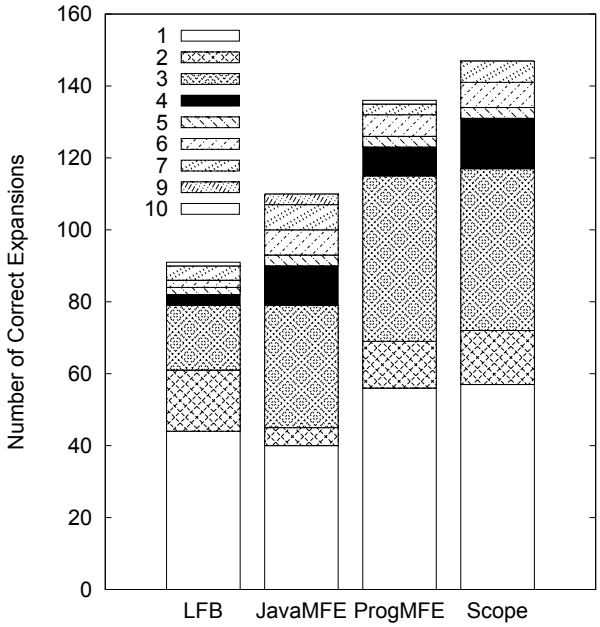

Figure 2: Number of correct expansions by abbreviation length in letters.

Figure 2 shows the number of correct matches broken down by short form length rather than type. Interestingly, LFB performed best on short forms of length 1 and 2 (with accuracy of 65% and 45%, respectively), worst on short forms of length 4 (only identifying 12% correctly), and average for longer expansions. As expected, the Scope approach had the highest accuracy for short forms of length 3 (64% accuracy), as well longer short forms of length 6 and 7 (78% and 60% accuracy, respectively). The MFE approaches perform similarly, although JavaMFE outperforms ProgMFE for longer length short forms and JavaMFE underperforms all techniques for length 2.

6. DISCUSSION AND FUTURE WORK

The results of our experiment demonstrate that our scoped approach is a significant improvement over the state of the art LFB. For the 227 NCW non-dictionary words, our technique had an accuracy of 63.0%, whereas LFB correctly identified just 91 long forms for an accuracy of 40.1%. Out of the 159 incorrect expansions for LFB, 123 short forms had more than one long form candidate and were therefore missed. Thus, the majority of long forms missed by LFB were due to not choosing between multiple possible expansions.

Despite our success over LFB, there is still room for improvement. We manually investigated the short form expansions missed by the Scope approach and identified some patterns in the set of missed and incorrect expansions which point to avenues for further research.

Humans unable to identify the long form. For some incorrect results, a human was unable to identify the long form. Out of 103 total incorrect expansions for our Scope approach, 36 were single letter abbreviations. Over half of the incorrect single letter expansions lacked a clear expansion given the context according to the human annotators. Some single letter abbreviations, such as 'i' or 'x', are commonly used out of convenience and add no semantic value to the code. In future, we plan to automatically identify when an abbreviation has no intended meaning rather than attempting to assign meaning where none is intended.

Incorrectly choosing between multiple candidates. A second class of missed short form expansions were due to selecting the incorrect candidate from multiple choices. For instance, the short form 'loc' was expanded to 'locate' instead of 'local'. The incorrect long form was found at the program level using the program MFE list because no expansion of 'loc' was found in the method. Both of these long forms are common within the program and occur in the same proximity. By refining our long form selection algorithm, especially at class and program levels, we hope to obtain more appropriate long forms when choosing between two seemingly acceptable long forms.

Long form based on domain knowledge absent in code. Missing domain knowledge also presents a problem to expanding abbreviations. For example, the short form 'lsup' occurred in the math-based typesetting class StarMathConverter, and no long form candidate was found in the entire program. The human annotator investigated beyond the program into domain knowledge and found that 'lsup' is a Tex command that stands for 'left superscript'. This particular short form was part of a set of abbreviations that are so common in math-based typesetting code that they are generally understood by the developers and thus long forms are not present anywhere in the code. Many domains have similar sets of generally understood abbreviations that make finding expansions exceedingly difficult. To remedy this problem, we plan to improve our Java MFE list by mining over more programs and hand-tuning the long forms for the most frequently occurring short forms.

Further improvements. The Scope approach could be further improved by using a specialized edit distance to eliminate unlikely dropped letter and combination word expansions. An edit distance [7] assigns a similarity score between two strings based on the number of edits, in terms of additions and deletions, required to convert one string, such as a short form, into the other, such as a long form candidate. For example, vowels are more likely to be dropped than consonants, so a specialized edit distance would penalize consonant additions more than vowel additions.

Another avenue of future research is in using a long form candidate's part of speech to eliminate unlikely candidates. For example, many abbreviation expansions in our gold set are nouns, such as 'integer' or 'string', or noun phrases, such as 'extensible markup language' or 'pseudo random number generator'. However, there are two issues that must be overcome with this approach. First, part of speech is more difficult to determine for software words than for English text. This is because many of the sentences and phrases in software are in the imperative form, which existing part of speech taggers are not trained on. Second, the part of speech of a short form, and thus its long form candidate, may vary depending on the location. Consider the short form 'def' in the identifiers defFont and fontDef. In defFont the appropriate long form is the verb 'define', whereas in fontDef the appropriate long form is the noun 'definition'. Thus, the location of a short form within an identifier must be taken into account with the parts of speech of the surrounding words.

Finally, in this work we have only begun to expand abbreviations for Java programs predominantly written in English. Although the technique can theoretically be applied to any natural language, further evaluation and development are necessary to maximize the performance of our automatic abbreviation expansion technique for other languages.

7. CONCLUSION

Automatically generated abbreviation expansions can be used to enhance software maintenance tools that utilize natural language information, such as search and program comprehension tools. In this paper we present an automatic mining technique to expand abbreviations in source code. Our scoped approach uses contextual information at the method, program, and general Java level to automatically select the most appropriate expansion for a given abbreviation.

We evaluated our approach on a set of 250 potential abbreviations and found that our scoped approach provides a 57% improvement in accuracy over the current state of the art [9]. In addition, we noted that applying the most frequent expansion (MFE) component of our approach at the program and Java level also provided improvements beyond the current state of the art.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments on this work.

8. REFERENCES

[1] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Trans. Soft. Eng.*, 28(10):970–983, 2002.

[2] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proc. Inter. Conf. Soft. Eng.*, 2006.

[3] B. Caprile and P. Tonella. Restructuring program identifier names. In *Proc. Inter. Conf. Soft. Maintenance*, 2000.

[4] H. Feild, D. Binkley, and D. Lawrie. An empirical comparison of techniques for extracting concept abbreviations from identifiers. In *Proc. Inter. Conf. Soft. Eng. and Applications*, 2006.

[5] F. Feng and W. B. Croft. Probabilistic techniques for phrase extraction. *Inf. Process. Manage.*, 37(2):199–220, 2001.

[6] E. Hill, L. Pollock, and K. Vijay-Shanker. Exploring the neighborhood with Dora to expedite software maintenance. In *Proc. Inter. Conf. Auto. Soft. Eng.*, 2007.

[7] D. Jurafsky and J. H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice Hall PTR, 2000.

[8] L. S. Larkey, P. Ogilvie, M. A. Price, and B. Tamilio. Acrophile: an automated acronym extractor and server. In *Proc. Conf. Digital Libraries*, 2000.

[9] D. Lawrie, H. Feild, and D. Binkley. Extracting meaning from abbreviated identifiers. In *Proc. Inter. Working Conf. Source Code Analysis and Manipulation*, 2007.

[10] B. Liblit, A. Begel, and E. Sweeser. Cognitive perspectives on the role of naming in computer programs. In *Proc. Annual Psychology Programming Workshop*, 2006.

[11] C. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, 1999.

[12] A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proc. Inter. Conf. Soft. Eng.*, 2003.

[13] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic. An information retrieval approach to concept location in source code. In *Proc. Working Conf. Reverse Eng.*, 2004.

[14] S. Pakhomov. Semi-supervised maximum entropy based approach to acronym and abbreviation normalization in medical texts. In *Proc. Association for Computational Linguistics*, 2001.

[15] M. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[16] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of duplicate defect reports using natural language processing. In *Proc. Inter. Conf. Soft. Eng.*, 2007.

[17] D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *Proc. Inter. Conf. Aspect-oriented Soft. Devel.*, 2007.

[18] C. Simonyi. Hungarian notation. In *Visual Studio 6.0 Technical Articles*. Microsoft Corporation. Reprinted 1999.

[19] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang. SNIAFL: Towards a static non-interactive approach to feature location. *ACM Trans. Soft. Eng. and Methodology*, 15(2):195–226, 2006.