IMPROVING AUTOMATIC ABBREVIATION EXPANSION WITHIN SOURCE CODE TO AID IN PROGRAM SEARCH TOOLS

by

Zachary P. Fry

A thesis submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Bachelor of Science in Computer and Information Science

Spring 2008

IMPROVING AUTOMATIC ABBREVIATION EXPANSION WITHIN SOURCE CODE TO AID IN PROGRAM SEARCH TOOLS

by

Zachary P. Fry

Approved: _____

Vijay K. Shanker, Ph.D.
Professor in charge of thesis on behalf of the Advisory Committee

Approved: _____

Lori Pollock, Ph.D.
Committee member from the Department of Computer and Information Sciences

Approved: _____

James L. Glancey, Ph.D.
Committee member from the Board of Senior Thesis Readers

Approved: _____

John Courtright, Ph.D.
Director, University Honors Program

ACKNOWLEDGEMENTS

First I would like to thank Dr. Lori Pollock and Dr. K. Vijay Shanker for years of support and advisement in my research efforts at the University of Delaware. I will carry everything I have learned throughout my carear.

I would also like to thank Dr. David Shepherd and Emily Hill for all of the help and mentoring they have both given me throughout my time researching with the Hiperspace group. Thank you for all the work you have done to get me to this point and for affording me all of these opportunities.

Finally I would like to thank my family and friends for helping me through all of my years at the University of Delaware and for all the fun we have had along the way.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Software maintenance is an important part of the software lifecycle. Understanding large software systems that are unfamiliar can be difficult for maintenance programmers. Intelligent and robust search tools are one method for facilitating program understanding and comprehension. One of the major problems associated with improving search tools is the use of abbreviations within software.

The focus of this thesis is proposing and evaluating an automatic approach to expanding abbreviations in search tools. The results of our evaluation demonstrated that the first implementation of our approach made significant improvement over no abbreviation expansion with respect to search tools. Furthermore, the improved implementation returned even better results qualitatively than the initial approach.

Chapter 1

INTRODUCTION

Abbreviations are found throughout many modern software systems. An increase in the size and scope of programs coupled with a general trend of more descriptive language constructs has led to an increase in the use of abbreviations. The purpose of abbreviations is to shorten code and increase efficiency throughout the development process. Unfortunately using abbreviations makes software maintenance more difficult because reading, understanding, and searching the software is impacted negatively.

In previous work [10] we developed an automated approach to mine abbreviation expansions from the software itself. We first classified abbreviations based on the type of abbreviation and the local context of the abbreviation occurrences. The hypothesis was that by examining more frequently occurring abbreviation types starting with the most narrow context and moving outward we would find the most suitable expansion for the abbreviation in question. Evaluation of our tool revealed that we made a significant improvement in accuracy over the state of the art abbreviation expansion approach.

While I participated in the initial development and first implementation of the automatic abbreviation expansion strategy, this thesis makes the following contributions:

- Analyzed the weaknesses of the initial automatic abbreviation expansion strategy

- Developed a set of modifications and refinements to the initial expansion algorithm
- Designed and conducted a significant evaluation of the refined approach within the context of a search tool

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

This thesis is organized into chapters based on my contributions. Chapter two motivates the problem of expanding abbreviations. In chapter three I present the background information on abbreviations and the state of the art in terms of automatic expansion. Chapter four outlines our preliminary approach and the subsequent evaluation of our technique. My investigation of the initial algorithm and proposed improvements are described in chapter five. The final two chapters detail the two evaluations of the refined technique in addition to the associated results and conclusions.

Chapter 2

MOTIVATION

2.1 Maintenance Tasks in Complex Software Systems

Software systems today are becoming increasingly large and complex. Additionally, most large software systems are programmed in popular Object Oriented languages like Java and C++ due to ease of scalability and organization. The drawback of these languages is that related code and information can get easily scattered across the entire code base and become very difficult to find.

Throughout the life cycle of an application, between 60-90% of resources are devoted to modifying the application to meet new requirements and to fix discovered faults [6]. These tasks are commonly referred to as maintenance tasks and, because of their frequency, are studied often with respect to possible increases in efficiency.

One of the many challenges posed by maintenance tasks is the diversity of resources associated with them. The developers that are completing the revision or addition may not be the same people who initially wrote the code. Additionally, many developers might be working on the same maintenance task concurrently. Similarly, lots of diverse code from different sources may be involved in completing just one task. This large range of possible inputs and resources makes organizing, structuring, and carrying out routine maintenance tasks difficult.

Thus, techniques for expanding abbreviations can improve the effectiveness of a variety of language-based software tools such as concern location [11, 16, 20, 21], documentation to source code traceability [2, 14], or other software artifact

analyses [3, 19]. Automatically expanding abbreviations will give these tools access to words and associated meanings that were previously meaningless sequences of characters.

The large, scattered, and semantically complex nature of software systems today necessitates more intelligent software tools. The most basic software tool is a simple search tool who's functionality includes matching a query with relevant code in the target software system, by finding code with idenifier names matching the query. However, as maintenance tasks evolve, so do the needs of the developer. Where once searching with a single term was sufficient, now developers desire the ability to look deeper into source code and extract information that would be otherwise hidden. For instance, instead of searching mere occurrences of terms, developers now want to find entire concepts in code. Basic searching needs to be expanded to meet these evolving needs.

With the prevalence of maintenance tasks and the increased need for faster development in today's rapidly changing economy comes a need for the aforementioned intelligent searching. With this capability, developers who are unfamiliar with the code base they are working with can quickly and easily navigate to the concepts they are looking for in code and therefore complete the task more quickly.

2.2 Challenges Posed by Coding Practices

Search capability depends entirely on the transparency and specificity of the code base. However, the nature of programming languages today allows developers to choose identifier names with very little restriction. For example, identifiers in Java can be any combination of upper and lower case letters, numbers, and select punctuation. Acceptable names would include a, temp_charArray, and c_8intB4. While abbreviation in software can save time and effort, it often leads to code that is difficult to understand if descriptive and informative naming conventions are not

followed. Similar difficulties can also arise in code where different terms are used in different places to mean similar concepts.

Nondescriptive or incomplete identifier names occur for several reasons. One cause is the high frequency that some words are used in programs. The words integer and character, for instance, are used with such frequency that most conventional languages use shorter forms in the defined syntax. In this case, "int" and "char" are substituted respectively for "integer" and "character" in the program keywords for popular languages such as Java, C++, and C among others. Programmers use many other words so frequently throughout programs that they have common substitutes that are used almost entirely synonymously throughout code. Examples include "string", "object", and "position" which are very frequently shortened to "str", "obj", and "pos", respectively throughout the computer science domain.

Another cause of nondescriptive naming of identifiers is the increased frequency of long identfier names in large, complex systems. In most object oriented languages, inheritance and polymorphism lead to large numbers of similar objects. Attempting to find unique names for increasingly similar objects often leads to long and tedious identifiers. Examples of such names in the Azureus [1] open source project are SecureMessageServiceClientMessageImpl and AzureusCoreSingleInstanceClient. In this case, each class name is at least 5 words long which is necessary for descriptive purposes but would be inefficient and bothersome to write often in the process of software development. For this reason, developers often find ways to shorten these names into more efficient but less generally recognizable forms.

In general, all of the techniques used by programmers to make identifier names more efficient and usable throughout code can be classified under abbreviations. Strictly speaking, the Merriam-Webster definition of an abbreviation is a shortened form of a written word or phrase used in place of the whole. [17] In software, abbreviations are essentially concepts applied to words and phrases in

```
dht = (DHTPlugin)dht_pi.getPlugin();
Thread t = new AEThread( "DHTTrackerPlugin:init" )
{
  public void runSupport()
  {
    try{
      if ( dht.isEnabled()){
        log.log( "DDB Available" );
        ...
      }
    }catch( Throwable e ){
      log.log( "DDB Failed", e );
    }
    ...
  }
}
```

Figure 2.1: Sample Code Demonstrating Abbreviation Use

program identifiers.

The frequency of abbreviations used throughout software is staggering. Figure 2.1 shows an excerpt of code from the open source project called Azureus in which abbreviations are present. A large subset of the identifiers within this code are abbreviations. Specifically, "dht", "t", "AE", "DDB", and "e" all occur in this small code snippet. In the five test programs used by Hill et al. [10] to evaluate an expansion technique approximately 5.7% of all the words within the code are non-dictionary words. While this may seem like a small percentage, in all this constituted 333,305 identifiers.

2.3 Abbreviations in the Context of Search Tools

The abundance of abbreviations in source code suggests that high quality search tools should adequately account for the inherent meanings. For instance, in

Figure 2.1, dht in this context actually stands for "distributed hash table". It is conceivable that a developer would want to search for the functionality associated with distributed hash tables but is unaware that the phrase was commonly abbreviated in the system as "dht". In this case, searching for "distributed hash table" would fail to return all code where the concept was represented by "dht". The highly frequent use of abbreviations in software motivates finding an accurate way to expand or match abbreviations to their intended meanings, leading to more intelligent and more effective search tools.

Chapter 3

AUTOMATICALLY MINING LONG FORMS

Identifying the long forms that correspond to an abbreviation as a human can sometimes be a challenging task. Automatically identifying them is even more difficult due to variations in abbreviation conventions and incomplete specification of the intended long form. This section discusses the basics of the abbreviation problem and the state of the art technique for expanding short forms.

3.1 Different Types of Non-Dictionary Words

Abbreviations are a subset of the non-dictionary words found in software. In a previous paper [10], we defined a classification system for non-dictionary words for the purpose of performing automatic abbreviation expansion. The following sections outline the different classes of non-dictionary words.

3.1.1 Single Word

Single word abbreviations are the most common type of abbreviation throughout software. There are two main subsets in this category: prefixes and dropped letter abbreviations. A *prefix* is a short form that is created by removing letters from the trailing end of a long form, leaving at least one letter in the beginning of the abbreviation. Examples of prefixes include "int" for integer and "str" for string. Single letter abbreviations comprise an important subset of prefixes. A common example of a single letter abbreviation is "e" for exception. This subclass demands

a unique approach to expansion that will be discussed in a later section on improvements to the original expansion algorithm. *Dropped letter* abbreviations are formed by dropping letters from anywhere inside the long form where there are at least two consecutive letters in the short form that are non-consecutive in the long form. Examples of dropped letter abbreviations are "evt" for event and "amt" for amount.

3.1.2 Multiple Word

There are two types of multiple word (or multiword) abbreviations: *acronyms* and *combination multiword* abbreviations. An *acronym* is formed by taking the first letters of multiple words and combining them into a single short form. Examples include "FBI" for "Federal Bureau of Investigation" and "ast" for "abstract syntax tree". *Combination multiword* abbreviations occur when at least one of a group of words is abbreviated by one of the aforementioned single word methods and then the words and/or abbreviations are combined. Some examples of this type of abbreviation are "println" for "print line" and "strcat" for "string concatenation". Note that in the first example, only one word is abbreviated while in the second, both words are shortened.

3.1.3 Domain Keywords and Special Cases

The final form of non-dictionary words consists of those words that are neither found in the dictionary nor follow any of the abbreviation constructs and are therefore not expandable. In general, these words tend to be individual special cases, but a few distinct classifications can be made. *Domain keywords* comprise a large portion of the non-dictionary special case words. This category includes words that are formally defined for a specific domain or doctrine as well as words that have grown out of practices associated with individual areas of work. Generally these words are derived from two or more dictionary words that are not split in a distinct enough manner that they could be automatically separated. Examples

of these words include "parsetree", "keystroke", and "serialize". A second classification of special cases includes *misspellings* and *arbitrarily chosen meaningless identifier names*. Spelling mistakes occur with surprisingly high relative frequency and cannot be matched to dictionary words with high enough certainty to correct them automatically. Developers also sometimes take the liberty of designing completely meaningless identifier names that do not lend themselves to automatically extracting meaning. We have chosen to only expand abbreviations because in most situations extracting further meaning from these special cases is either too difficult or pointless.

We identified several additional challenges associated with expanding long forms, including poor dictionaries, difficulty identifying short form types, and the difficulty of choosing between multiple long form candidates. [10] For this reason, the problem is not a trivial one.

3.2 State of the Art in Abbreviation Expansion for Software

Lawrie, Feild, and Binkley are the only other researchers to present and evaluate techniques to address the problem of automatically expanding abbreviations that occur in program identifiers. [13] In their earlier paper, 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. [7] 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) 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. [13] 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 markers (e.g., camel-casing) occurring in f, and a phrase dictionary created by running the comments and multi word-identifiers through a phrase finder [8]. Stemming and the stop word list are used during extraction to improve accuracy. The first letter of each phrase is used to build acronyms. In addition to the lists for each function, they create a list of programming language specific words as a stop word list. Expansion of a given non dictionary word occurrence in a function f involves first looking in fs 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 [13] 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 and 64% of the over-two letter forms were expanded correctly. 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 [4]. 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 [12, 18]; 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.

We identified two major problems that were unaddressed by the state of the art techniques. First, improved precision is needed for the abbreviation expansion technique when identifying possible long forms. Secondly, a technique is needed for choosing between several possible expansions are found in the same context. Our scoped approach attempts to address these fundamental problems.

Chapter 4

A SCOPED APPROACH TO AUTOMATIC ABBREVIATION EXPANSION

Our goal in automating a system to extract abbreviation expansions was to be as accurate as possible. The state of the art techniques worked to a point but failed to choose between multiple expansions when they occurred, which we found to be very frequently. This problem, unfortunately accounts for the majority of the difficulty associated with abbreviation expansion. Our method is called the *scoped approach* because we aim to identify the most likely short form expansion by starting at a narrow context(which we call scope) and limited classification and expanding outwards until a match is identified. We developed an algorithm for choosing an expansion if more than one possibility is identified at a given level of scope. Finally, if no expansion was identified we default to the *Most Frequent Expansion* (MFE) based on other occurrences of the short form. Our initial implementation was in the form of a Java Eclipse plugin and command line scripts for the MFE calculations. A detailed description of this approach and a formal outline of the algorithm can be found in paper published earlier [10]. This chapter presents an overview of this technique.

4.1 The Scoped Approach

The key approach is to begin with a very narrow scope and expand outwards while inspecting for possible long forms and making an intelligent choice where

multiple expansion possibilities occur. The scope in this case is in terms of both abbreviation type and program context.

4.1.1 Abbreviation Type

The scoped approach begins by creating a regular expression for the short form in question with respect to attempting to match the different types of abbreviations. Based on inspecting many systems to observe which types of abbreviations occurred most frequently and in what contexts, we determined a set order for checking the abbreviation types. The general order is to begin with acronyms, then check for prefixes, dropped letters, and multiple word expansions in that order. Again, this ordering was based on specificity of the type in addition to the frequency with which it accounted for a correct expansion based on manual inspection of code. More comprehensive justification for this process can be found in the previous paper [10].

4.1.2 Program Context

The process of searching for each abbreviation type individually was guided by the short form's program context. The algorithm checks the enclosing method's Javadoc comments if any are present. Javadoc comments take a very specific form whereby an identifier is listed followed by description. We found that, if present where the short form in question was the identifier specified by the comment, any possible expansions found were overwhelmingly correct. Failing that, if the abbreviation is an object, the type is inspected to identify possible expansions. From this point, the statement and then method body are checked. Finally, if no match is made, the method comments and then class comments are examined. If no possible expansions are found in any context for the given abbreviation type, this process is abandoned and the process is repeated with the next abbreviation type in the ordering.

4.1.3 Multiple Matches

Frequently, multiple matches occur within a single scope. We designed an algorithm to intelligently handle this situation by trying to automatically choose the most likely candidate based on several metrics. We first examine the frequencies with which each potential long form occurs in the given scope. If the long forms for the same short forms all occur with the same frequency, we stem the long forms and recalculate the frequencies. If still no winner is chosen, we broaden the contextual scope, again reexamining frequency. If still no distinction can be made, we rely on the most frequent expansion (MFE) technique outlined in the next section.

4.2 Most Frequent Expansion (MFE)

The last possible approach to try to expand an abbreviation in code is what we refer to as using the Most Frequent Expansion (MFE). If no discernable long form can be found inside the containing code base, our hypothesis is that perhaps trying other code bases might provide supplementary information.

We create a list of short forms and their most frequent expansions by running our abbreviation extraction approach on the Java 5 code base and by keeping a count of the frequencies in which each long form is matched. We chose the Java code base due to its relatively large size (1.5 million lines of code) and generally standard coding paradigms, though the MFE could be calculated using any code base. It should be noted that this approach is indeed a last resort and that, though this approach is at best an educated guess, we felt that it was better than simply returning nothing at all.

4.3 Evaluation and Results

After implementing the method developed by Lawrie et al.(cite) as a means of comparison for our approach, we ran an experiment to compare the accuracy of both strategies. We found that overall our method performed 57% more accurately

than the state of the art. Another metric included the effectiveness of only the Java MFE as compared with an MFE calculated by using the subject program. While promising, these results showed ample room for improvement.

Chapter 5

ANALYSIS AND REFINEMENT

5.1 Analysis of Expansion Algorithm Weaknesses

An initial inspection of the experimental results of the scope approach to automatic abbreviation expansion showed several key problems overlooked by our first approach. Our initial evaluation used a gold set of 250 abbreviations of which we expanded 102 incorrectly. Starting with this subset I attempted to classify the incorrect expansions into similar and identifiable problems. Three major issues were identified within our initial scope abbreviation expansion algorithm along with several smaller ungeneralizable discrepancies.

The first problem that arose concerns false positives based on abbreviation type which brought into question the validity of exhausting every program context for one abbreviation type before checking even the narrowest scope for another. An example of this type of incorrect expansion by our scope strategy, which we will call *Order of Scoping*, was found in Azureus [1] with respect to the abbreviation "dl" and is as follows:

```
Download plugin_dl = PluginCoreUtils.wrap( download_manager );

...

// due to latencies we need to give speed increases a time to take
```

- Actual Expansion: "due latencies" based on checking acronym in the class-comment scope

- Correct Expansion: “download” which is a Dropped Letter abbreviation on the statement level

Because the initial scope strategy checked for acronyms on every context level before checking for other types of possible expansions, the fact that “due” and “latencies” coincidentally appeared in a completely unrelated comment in the same class led to an incorrect expansion.

A second major problem with our initial implementation concerns the fact that single letter abbreviations were expanded correctly much less than longer short forms. The frequency of occurrence of single letter abbreviations motivated looking deeper into the cause of this inaccuracy. An example of incorrect *Single Letter* expansion can be found below. In this case, many meaningless single letters appear and a possible false expansion could occur with the abbreviation “d” being incorrectly expanded to “display” at the method level.

```
int[][] matrix = new int[10][5];
for(int i=0; i<10; i++) {
  for(int j=0; j<5; j++) {
    int a=i, b=j, d=10;
    updateDisplay(display);
    matrix[i][j] = a*b*d;
  }
}
```

Finally, it appeared that there was a rather small set of words that were abbreviated so often in general software that their corresponding long forms often do not occur in the code thereby making them difficult to expand automatically. The previous implementation of our abbreviation expansion technique addressed this by checking all nondictionary words that were unmatched at the end of the expansion process against a list of common abbreviations. However, in the aforementioned gold set there was a case as follows:

```
// then clone the sub-tree recursively
```

```
...

public static Node getNodeInfo(Node node) {

        char[] str = null;
        switch (node.getNodeType()) {

         case Node.ELEMENT_NODE:
             str = "ELEMENT";
             break;

         ...
```

- Actual Expansion: "sub tree recursively" based on checking acronym in the class-comment scope
- Correct Expansion: "string" which is a prefix abbreviation on the type level

In this example, because we check for common abbreviations at the very end of the expansion process, str gets expanded incorrectly because of a coincidental class comment. After further inspection, I noticed that this type of incorrect expansion happened surprisingly often because of the relatively high possibility that a series of words in a larger scope could coincidentally be used as a potential expansion of a nondictionary word.

5.2 Order of Scoping

The previous strategy of the scoped method started with an abbreviation type and then checked every level of context for that type, only checking the next type when all possible contexts were exhausted. The general idea of context is that the broader the context, the more unlikely a potential long form is going to be the correct expansion. In other words, the possibility that a long form matches by coincidence only is greatly increased when the contextual scope is broadened. Comparatively, the order with which we check the types of abbreviations shows

much less variation in potential to expand short forms correctly throughout the entire hierarchy. While the order is firmly grounded by observational conclusions, the difference in accuracy between the levels of contextual scope is much greater. Therefore, it would follow that starting with the most narrow level of contextual scope and exhaustively testing every abbreviation type before trying the next highest context would yield more accurate results.

5.3 Single Letter Abbreviations

Out of all the non-dictionary words in the five test programs used by Hill et al. 28.9% are single letter abbreviations. As this is essentially the most time saving and efficient way to abbreviate, it would naturally follow that single letter abbreviations would occur with such frequency throughout code. The problem associated with this kind of abbreviation is that it generally leaves the least amount of evidence as to what the corresponding long form is. This is not to say, however, that finding possible expansions is difficult. On the contrary, because this special form is treated, by default, as a prefix any word found in the code that begins with the letter in questions is a potential match. Because of the extreme greediness associated with this approach, choosing the correct expansion out of a large pool of possibilities given limited clues as to what the original developer intended proves to be a very difficult problem.

Yet another pitfall of single letter abbreviation is the fact that commonly they are used as default variable names that were never intended to have either a long form or a meaning at all. Take, for example, the code illustrating single letter abbreviations in section 5.1. It is entirely conceivable that code of this nature could be found in many systems. Temporary variables and local variables with a very small scope are often named with meaningless letters out of convenience. With loop conditionals, it is convention to use single letters as identifiers. Logically, if

these identifiers had no intended long form, they arguably do not even qualify as abbreviations and thus it does not make sense to try to expand them.

This conclusion led me to devise a new scheme for handling single letters. Initially, we assume the single letter is an abbreviation and begin by looking at contextual scope as per the original approach. However, only Javadoc comments and type are examined because after close inspection it appears that broadening the context beyond this point generally yielded more false positives than correct results. If no match is found beyond the type context, we deem the letter to have no meaning and cease the expansion process. While it may seem that this may lead to overlooking some letters that are indeed actually abbreviations, we found that this subset of incorrectly identified meaningless identifiers was far smaller than the previous subset of false positives that were a product of trying to expand false abbreviations. A more detailed description of the increase in accuracy can be found in the evaluation sections that follow.

5.4 Hyper-Common Abbreviations

Some words occur so frequently in code that abbreviations of these words have not only become synonymous with the words themselves but, in many cases, have completely replaced the word altogether. I have identified this small subset of words as *hyper-common abbreviations*. A list of the subset I have identfied for our purposes can be found in Table 5.1. This subset was compiled by first sorting a list of every abbreviation in the Java 5 code base by frequency of occurrence. Then, of those that occurred very frequently, any that were expanded to a single long form more than 90% of the time were added to the list. It seems very unlikely, for example, that one would be able to find many instances of the abbreviation "int" in code where the intended long form, "integer", occurred as well. It is generally understood that the meaning of "int" is integer and therefore specifying the long form in the code would be redundant. For this reason, automatically extracting

long forms for these words with the scoped method is extremely difficult. However, the fact that they are indeed so common and are generally meant to be expanded to a unique long form provides a suitable solution. Our method for identifying and expanding these abbreviations is to attempt to match every abbreviation to the list of hyper-common abbreviations before trying the scoped approach and, if it is indeed on the list, expanding it to the corresponding long form. We feel that the small chance that one of these words could perhaps be an abbreviation for something else and therefore automatically matching it out of context would result in an incorrect long form is negligible compared to the increase in accuracy due to the reduction of incorrect expansions by the standard scoped method. Additionally, because of the frequency with which the words in this small subset occur within code, checking for them first reduces a significant amount of needless runtime in the overall process.

5.5 Inherent Inaccuracy

The remaining incorrect expansions could not be generalized into any concise categories and therefore no automated solution could be developed to increase accuracy any further. One such example concerning the abbreviation "ms" is as follows:

> Notice that "ms" appears in the formal parameter for the "login" method.
>
> ```
> public static void login(long maxDelayMS) {
>   ...
>   playAfterURL = (String) MapUtils.getMapString(reply,
>    "play-after-url", null);
>   ...
>   public void messageSent(PlatformMessage message) { ... }
>   ...
> }
> ```
>
> The only possible expansions for "ms" ("map string" and "message sent") both occur only once at the method level and were both of the

Short Form	Long Form	Accuracy
int	integer	82.1
impl	implement	84.0
obj	object	100.0
pos	position	82.8
init	initial	95.5
len	length	99.0
attr	attribute	100.0
num	number	98.5
env	environment	97.2
val	value	89.4
str	string	88.1
buf	buffer	99.2
ctx	context	96.2
msg	message	97.7
var	variable	97.4
param	parameter	99.2
decl	declare	92.0
arg	argument	96.4
rect	rectangle	96.7
expr	express	99.8

Table 5.1: Hyper-Common Abbreviations - Note: Accuracy in this sense refers to the number of times the given abbreviation was matched to the provided long form in the Java 5 code base.

acronym type. In this situation, both the previous and the current approaches would not be able to choose the most likely expansion. This is an example of where our automatic technique is insufficient to solve the automatic abbreviation expansion problem.

As is with any automatic procedure, there is going to be inherent inaccuracy. However, I feel that the current implementation is not only a significant improvement on the state of the art, but is suitable for the foreseeable applications.

Chapter 6

EVALUATION IN THE CONTEXT OF A SIMPLE SEARCH TOOL

We first evaluated the refined abbreviation expansion algorithm in the context of a simple search tool to answer the following research question:

1. When abbreviations are expanded in software, how many more search results are returned than without expansion?

Evaluation of the expansion technique inside of an existing software maintenance tool provides the most concrete evidence that it is indeed aiding in search tasks that would occur throughout the software development and maintenance cycle. Therefore, we designed this experiment to most accurately reflect the actual use of a common search tool with functionality similar to that of grep. The tool returns instances of the query terms that occur in the code being searched and therefore computing the percent increase in returned results is very straightforward.

The research question for this experiment deals with how many results were returned overall. In this case some queries may be searching for some words that are never abbreviated in code and some words that are. Any more returned results overall indicates that our technique aids in making search tools more accurate.

6.1 Experiment Design

6.1.1 Variables and Measures

The independent variable in this experiment is the abbreviation expansion technique, both original(called Scope) and revised(called iScope) strategies. The strategies are being evaluated by running search queries on both code containing only abbreviations and code that contains both the abbreviations and their automatically extracted long forms.

The dependent variable in this experiment is the effectiveness of the search tool when using our abbreviation expansion strategies and when no abbreviation expansion is used. This is measured over all queries in terms of percent increase in results. Results in this sense means the raw returned results. One instance where a query is found in source code is considered to be one returned result in the calculation of percent increase. We computed the percent increase in returned results from out strategies by dividing the number of raw results returned when abbreviation expansion is applied by the number of raw results returned when using the unannotated source and then subtracting 100%.

6.1.2 Subjects

Our subject queries for this experiment were developed from concerns outlined by Eaddy et al. [5] In their paper, "Do Crosscutting Concerns Cause Defects?", Eaddy et al located and verified 480 concerns for use in their work. For this experiment we narrowed the comprehensive list to a subset of 215 concerns based on size. In this thesis, any mention of the set of benchmark concerns refers to this subset. Concerns were deemed acceptable for our purposes if the associated code encompassed at least 10 methods and no more than 3 standard deviations above the mean concern size, which in this case was 112.5 methods.

Because this experiment's methodology requires queries and not concerns, we devised a method for translating the set of concerns into corresponding queries. We

identified 8 graduate and undergraduate students with Java programming experience to develop queries manually from the benchmark concerns. After verifying the human subjects' programming aptitude and qualification for this task by administering a brief survey, they were all asked to examine between 80 and 82 number of concerns. To mimic real world application as best as possible, we asked each subject to act as if they were attempting to search for the given concern within a standard search tool. Their task was to formulate a descriptive and appropriate search query for the given concept and associated concern within the code. Every concern was examined by exactly 3 different people to provide a comprehensive and representative range of possible queries. Each human subjects' individual queries were aggregated to form the set of subject queries for this experiment.

6.1.3 Methodology

The first step to evaluating the expansion process was to measure how many more results were returned over the set of 215 concern location queries when the abbreviations were expanded in the code. To do this, we first ran the query searches with the corresponding source code and recorded the returned results. Then the source code was annotated by running our initial abbreviation expansion technique and inserting the expansions at the abbreviation site whenever one was found. By running the query searches again on the annotated code, we gathered the returned results after abbreviation expansion. Similarly, the improved abbreviation expansion technique was applied and code annotated. Finally, percent increase in results was computed as described above.

6.2 Threats to Validity

As with any experiment that attempts to generalize results overall by examining a subset, there is an inherent threat in assuming that the subset will be representative of the entire domain. We address this issue by using a relatively large

test set of queries in hopes that the size will lead to more comprehensive coverage of all possible scenarios.

We chose Java as the language to investigate for our technique due to the large amount of available open source code. While it is only one of the many popular languages today, Java is certainly one of the most widely used and closely studied languages. Additionally, the same principles being studied with respect to Java could be generalized well to other similar object oriented languages.

Another possible threat to validity with this experiment is that while we measure the improvement in quantity of results, the actual quality of results is ignored. The second experiment described in this thesis focuses solely on the quality of results. The experiments were kept separate so the different dependant variables could be examined separately and a more accurate picture of the effectiveness of the abbreviation expansion techniques could be gained.

6.3 Results

Table 6.1 shows the results of this experiment. While searching after both abbreviation expansion strategies yielded more raw returned results than the original code without abbreviation expansion, the original abbreviation expansion strategy exceeded the improved abbreviation expansion strategy by a little over 3%. While this may seem to be a negative result, it should be noted that this is to be expected given the nature of the improvements made to the original abbreviation expansion strategy. Perhaps the most profound change was eliminating meaningless single letter abbreviations. Therefore, the slight decline in number of results returned can be attributed to my attempt to eliminate false positives associated with single letter abbreviations. While a full examination of the quality of the 1,671 results in question was not permitted given the time frame of this project it can be assumed that the majority of this decrease is due to single letter expansions. As mentioned in section 5.3, 28.9% of all abbreviations are single letter abbreviations. Therefore, because

Method	Total Returned Results	Percent Increase
No Expansion Method	240,752	—
Initial Implementation	284,160	18.03
Improved Implementation	282,489	17.34

Table 6.1: Results of Experiment 1 - Percent increase is measured over the number of results returned when no expansion technique is present

the discrepancy in results is only 0.69% between the two implementations and the percentage of single letter abbreviations is so high in comparison, it is conceivable that the overwhelming majority of the decrease in results is due solely to the removal of meaningless single letter abbreviations. An example of a result Scope could have returned that iScope would have not is as follows:

```
public void showTable(int actual) {
   string a = getEntry(0);
   int index = actual;
 ...
}
```

In this example code, if one of the query words in the experiment was "actual," then Scope would have expanded "a" to actual and returned "a" for the query containing actual when, in fact, it clearly has nothing to do with the query word. In comparison, iScope would have deemed "a" to be a meaningless single letter and not returned it for a query containing the word "actual". In this case, the extra result returned by the Scope method is actually a false positive which is why the slightly higher increase in returned results for Scope is negligible.

Furthermore, the goal of this experiment is to show that the improved strategy increases the aggregate results at all. A gain of 17.34% is significant in terms of improving accuracy and the number of overall results. A more detailed evaluation of the increase in quality of returned results can be found in the next experimental section.

Chapter 7

EVALUATION IN CONTEXT OF A CONCERN LOCATION TASK

We evaluated Scope and iScope in the context of software maintenance tasks to answer the following research question:

1. How much increase in effectiveness can be gained from expanding abbreviations in source code when performing concern location tasks?

As was stated in the first experiment, evaluation is most useful in the context of real world applications. Throughout the life cycle of an application, between 60-90% of resources are devoted to modifying the application to meet new requirements and to fix discovered faults [6]. To modify an application, developers must identify the high-level idea, or concept, to be changed and then locate (or find), comprehend, and modify the concept's concern, or implementation, in the code [15]. Concerns can often be scattered throughout the code base and thus locating them is a difficult and important task. For example, in the open source juke box application Jajuk [9], program code related to a music player resides in the abstract Player object (this includes the core player functionality of playing, pausing, stopping, muting, and seeking). This object-oriented decomposition makes comprehending and maintaining Player easier but can cause action-oriented concerns, such as the code related to playing a track, to become scattered across the system. Because concern location is so heavily involved in the software maintenance cycle, we designed this experiment to reflect the use of a common search tool to perform this task.

7.1 Experiment Design

7.1.1 Variables and Measures

The independent variable in this experiment is again the abbreviation expansion technique. We evaluated the original(Scope) and the revised(iScope) abbreviation expansion technique by measuring the increase in mean average precision when using code annotated with our technique, as opposed to not using any abbreviation expansion, inside two concern location tools based upon latent semantic indexing (LSI) and log entropy respectively. Without providing detail outside the scope of this thesis, LSI attempts to return the most relevant results by simply matching the query words within the documents and calculating similarity values based on the number and vicinity of occurrences. Similarly, the basic goal of log entropy is to reward long forms that co-occur as much as possible in the local vicinity of the short form but as little as possible in the global or non-local context of the short form.

The dependent variable in this experiment is the effectiveness of the expansion process in the context of a search tool when performing concern location. In this case, the ideal result set would include every method associated with a concern and exclude any method not appearing in the concern. Thus, we measure effectiveness in terms of mean average precision (MAP) with respect to attempting to locate concerns without abbreviation expansion versus the original and refined abbreviation expansion techniques.

Precision, a common information retrieval metric, is calculated by dividing the true positive results by the total number of expected positive results. MAP attempts to reward techniques that return relevant documents highly ranked. The first step to calculating a MAP value is to get precision measurements for every subset of size n where $1 < n <$ *size of result set*. Then by summing the precisions of only the subsets that end in a true positive and taking the average, a MAP value for the given query is calculated.

7.1.2 Subjects

This experiment uses the same subset of 215 concerns taken from the benchmark set of concerns developed by Eaddy et al. [5] used in the experiment outlined in Section 6. The corresponding set of queries we developed were also described in the section covering our initial experiment.

7.1.3 Methodology

The goal in evaluating the abbreviation expansion strategies in terms of a concern location task is to measure the difference in mean average precision between tasks over the set of 215 concern location tasks. To do this, we ran the concern location queries with both LSI-based and log entropy-based concern location tool over the source code annotated with expansions from both Scope and iScope and source code with no expansions present to get benchmark results. For the results corresponding to the concern location methods we calculated mean average precision for iScope compared to no expansion and for iScope in comparison with Scope. The method of comparison used was to take the difference between iScope and Scope and the difference between iScope and no expansion with respect to both concern location methods and plot the distribution of the differences.

7.2 Threats to Validity

The same three threats outlined in the chapter describing the first experiment I conducted also apply to this experiment. Specifically, these threats included generalizability of both our subject queries and the Java language in addition to evaluating, in this case, only the qualitative increase in the results. For explanation of my strategy in mitigating these threats please see the corresponding section in the previous chapter.

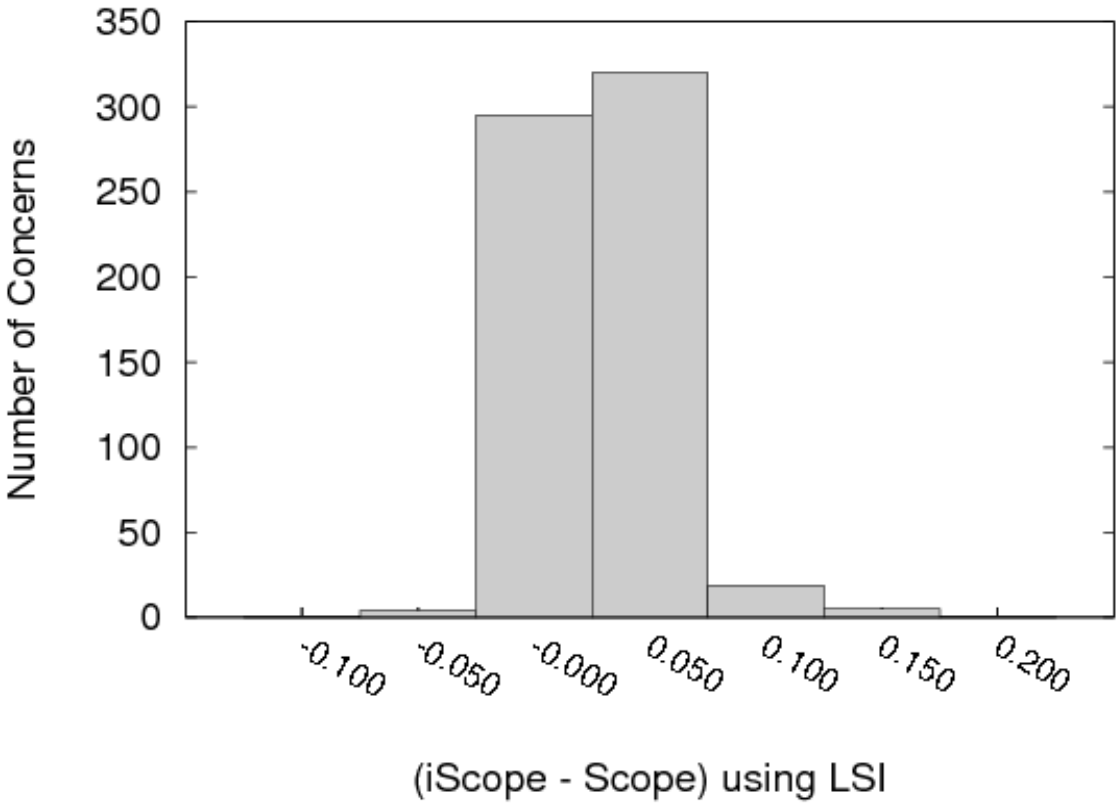

Figure 7.1: Distribution of the Differences in Mean Average Precision for iScope and Scope Using LSI

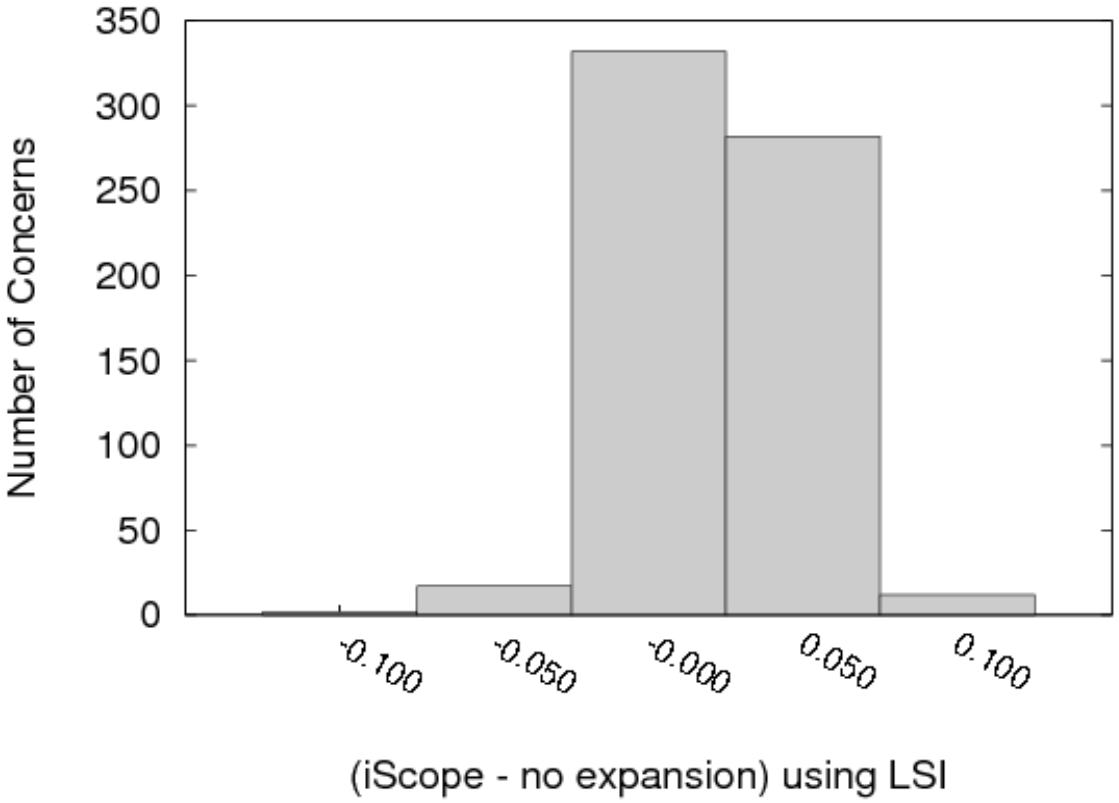

Figure 7.2: Distribution of the Differences in Mean Average Precision for iScope and No Expansion Using LSI

7.3 Results

The focus of this experiment was to evaluate our tool's effectiveness in facilitating concern location tasks. Each of the four plots resulting from this experiment measure the distribution in difference of MAP between iScope and either Scope or no expansion. In all cases, the x-axis corresponds to the difference in MAP between the two tasks(by subtracting either Scope or no expansion from iScope) while the y-axis corresponds to the number of MAP scores found for a given range. Note that a positive MAP value indicates that iScope outperformed the other technique in question. Also, each interval on the x-axis includes its corresponding label as an inclusive upper bound. With respect to LSI-based concern location, the iScope technique outperformed the Scope technique as shown in 7.1. However, as can be seen in figure 7.2, both iScope and Scope were outperformed by no abbreviation expansion whatsoever. This unexpected result can be attributed to the fact that LSI focuses on semantics of identifiers and therefore can circumvent the problem of abbreviations within code. Therefore, abbreviation expansion is not applicable for this technique. However, this is not to say that abbreviation expansion is not useful in other situations, as was proven by previous evaluations. The log entropy search method uses less semantic information than LSI. Therefore, log entropy is a more suitable context to test the abbreviation method because effectiveness can be tested more directly without additional semantic analysis clouding the results.

Figure 7.3 shows the mean average precisions when iScope was compared with Scope using the log entropy concern location method. Again, iScope outperforms Scope as was the case when LSI was used. However, as can be seen in figure 7.4, iScope also outperforms no expansion technique when using log entropy. As expected, the iScope abbreviation technique excelled in the context of a search method without semantic analysis because of its ability to augment the semantic information inside source code.

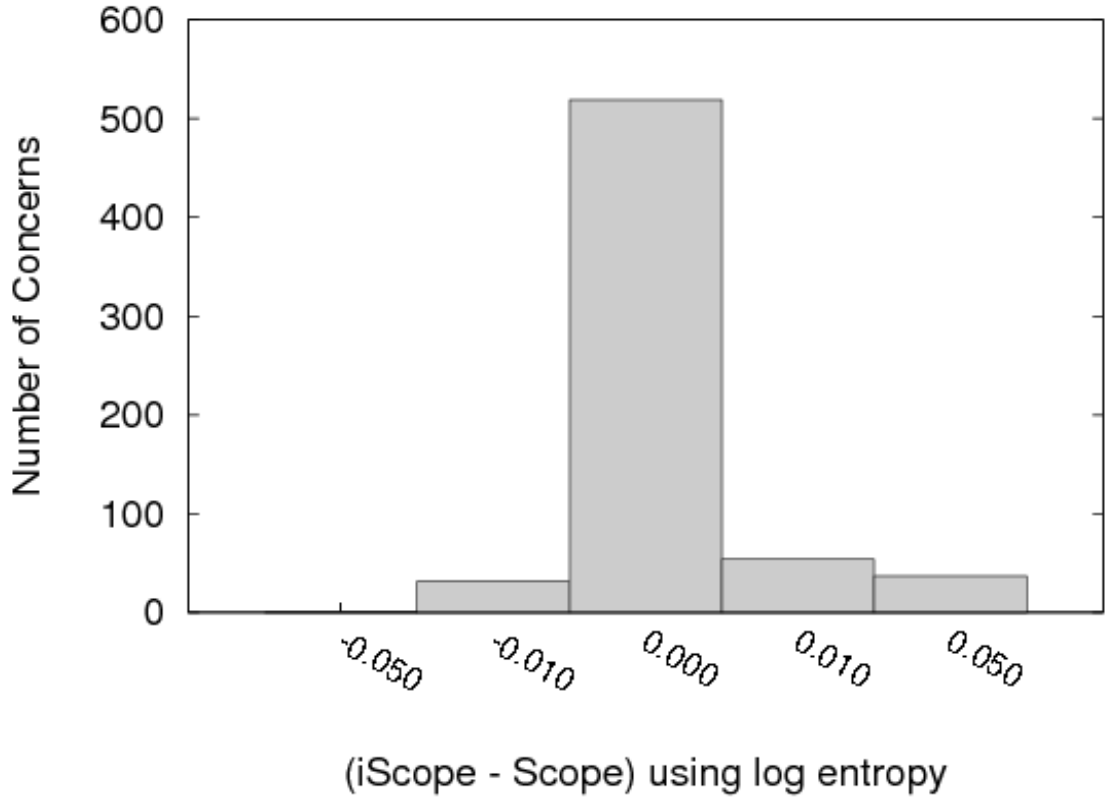

Figure 7.3: Distribution of the Differences in Mean Average Precision for iScope and Scope Using log entropy

Although abbreviation expansion proved to be unnecessary in the context of LSI-based concern location tasks, we felt the fact that iScope outperformed Scope in this context is meaningful in terms of showing iScope improved upon the Scope framework. Furthermore, the fact that iScope outperformed both Scope and no expansion in the context of log entropy-based concern location tasks both solidifies the proof of improvement over Scope and the effectiveness of abbreviation expansion as a concept to improve concern location tasks.

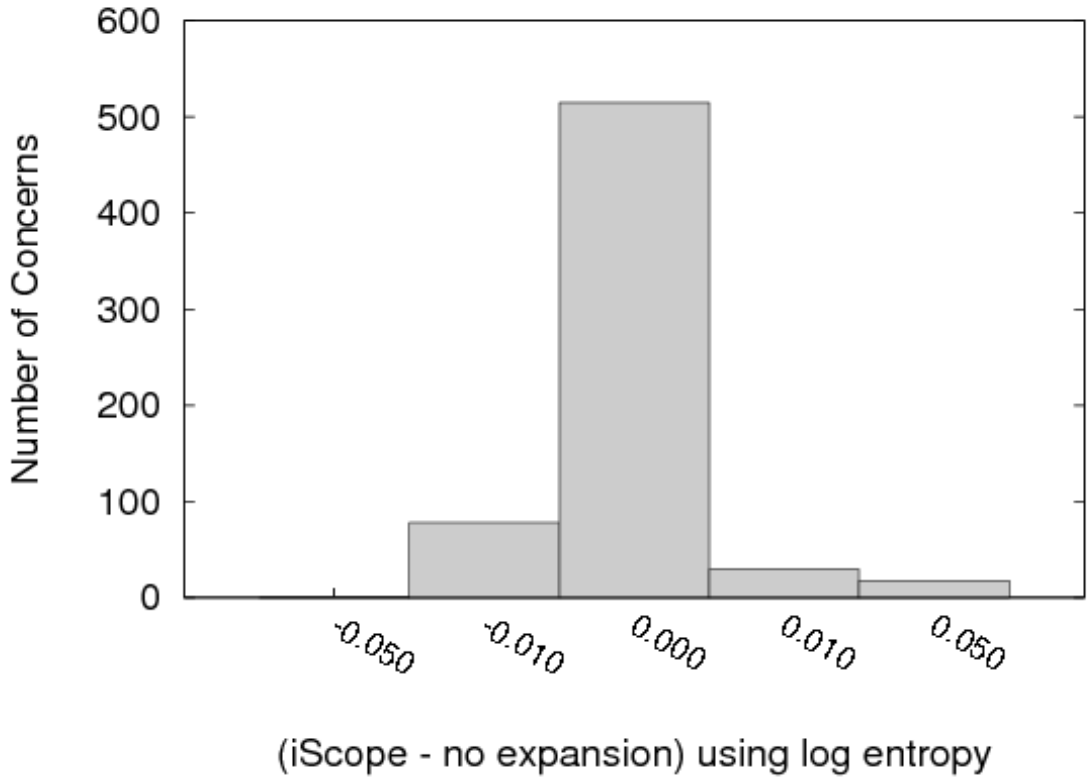

Figure 7.4: Distribution of the Differences in Mean Average Precision for iScope and No Expansion Using log entropy

Chapter 8

REEVALUATION IN THE CONTEXT OF A PREVIOUS EXPERIMENT

For a final evaluation of our refined expansion technique, we repeated the evaluation defined in a previous publication to answer the following research question:

1. Based on our previous experimental methodology and metrics, how much improvement was made from the initial strategy to the revised strategy?

8.1 Procedure

The independent variable in this experiment is the abbreviation expansion technique, both old and new strategies which, for the purposes of this experiment we will call *Scope* and *iScope* respectively. Our method is being evaluated by the same criteria as is outlined in the paper we published entitled: "AMAP: Automatically Mining Abbreviation Expansions in Programs to Enhance Software Maintenance Tools". The idea of this initial evaluation was to compare the accuracy of the Scope technique to that of using only the MFE(both Java and program level) approach discussed previously. Additionally, we examined the difference in accuracy between Scope and the state of the art technique by Lawrie et al. [13] which we refer to as LFB. One important detail is that due to our inability to implement the LFB strategy completely(specifically the combinational word expansion) because of a lack of information available, we compare our techniques with the LFB technique

excluding combination word abbreviations. This evaluation, like the previous one, measures accuracy by dividing the number of correct expansions over the number of total possible correct expansions.

The subjects in this and the previous experiment are 250 non-dictionary words taken from 5 large open-source programs. Each non-dictionary word was matched by human inspection to its correct expansion or deemed to have no match if none could be found or none was appropriate. We call this set of abbreviations and their respective expansions the "goldset." There was one minor change we made to the goldset when rerunning the experiment for this thesis which is described in the section below.

8.1.1 Modification of Goldset

Our previous experiment utilized a *goldset* of 250 randomly chosen non-dictionary words that were used to measure the accuracy of our approach. This set was comprised of short forms and their corresponding long forms based on human inspection of the associated code given instructions based on our assumptions. For the purposes of this experiment we slightly altered the *goldset* based on assumptions made in the new approach. Specifically, the idea of meaningless single letters extends to loop variants which, in the previous set, were deemed to have meaning. Therefore, for this experiment we changed every loop variant's long form to "no meaning". We feel that this change is warranted because the initial *goldset* was formed under the assumption that single letters have meaning which we have now deem to be incorrect. Therefore, we need to update the set of ideal expansions to reflect this insight.

8.2 Results

Table 8.1 shows the results for the reevaluation. As stated earlier, to compare both iScope and Scope to the LFB technique fairly, we use the non-combination

Type	LFB	JavaMFE	ProgMFE	Scope	iScope
CW	0.000	0.304	0.000	0.174	.391
DL	0.111	0.778	0.667	0.778	.778
OO	0.830	0.638	0.596	0.468	.851
AC	0.286	0.122	0.408	0.469	.429
PR	0.322	0.729	0.746	0.797	.593
SL	0.222	0.079	0.349	0.492	.587
NCW	**0.383**	0.401	0.529	**0.573**	**.617**
Total	0.348	**0.392**	**0.480**	**0.536**	**.596**

Table 8.1: Percent correct expansions for each technique and abbreviation type. CW: combination word DL: dropped letter OO: other(domain, meaningless, etc.) AC:

word(NCW) accuracy measures. For comparing iScope with Scope, Java MFE, and Program MFE we used the complete gold set accuracy totals at the bottom of table 8.1. All other values in table 8.1 correspond to each techniques accuracy with respect to individual abbreviation types. iScope improved by 6% over the previous Scope strategy in total and by 23.4% over the LFB state of the art approach.

In this case, the improvements made to the Scope technique provided a relatively small increase in accuracy however, overall I feel that an accuracy of almost 62% is sufficient for the application of improving program search tool accuracy. These results prove that our iScope expansion technique is accurate in terms of expanding abbreviations. This fact compounded with the conclusions that abbreviation expansion aids in program search and concern location tasks proves that our technique would be helpful facilitating software maintenance tasks.

Chapter 9

CONCLUSIONS AND FUTURE WORK

Automatic abbreviation expansion can aid software maintenance tools by exploiting the natural language and program structure clues inherent in software. Our initial Scoped approach provided a substantial increase in accuracy over the state of the art expansion techniques. Furthermore, my improved iScope technique enhanced the quality of the results returned in addition to accuracy. The final contribution of this thesis is a thorough evaluation of the effectiveness of our abbreviation expansion technique with respect to increase in quantity, increase in effectiveness when performing concern location tasks, and increase in accuracy.

In future work we hope to further refine the expansion process to achieve the most accurate expansions possible. Additionally we hope to fully integrate our abbreviation expansion process into a working program maintenance tool to fully evaluate its usefulness. Finally, we wish to extend the expansion process to additional programming languages to test its effectiveness across different domains.

BIBLIOGRAPHY

[1] Azureus: Bitorrent client. Online, 2008. `http://azureus.sourceforge.net`.

[2] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, 2002.

[3] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 361–370, New York, NY, USA, 2006. ACM Press.

[4] B. Caprile and P. Tonella. Restructuring program identifier names. In *ICSM '00: Proceedings of the International Conference on Software Maintenance (ICSM'00)*, page 97, Washington, DC, USA, 2000. IEEE Computer Society.

[5] M. Eaddy. ConcernTagger case study data. Online, 2008. `http://www1.cs.columbia.edu/~eaddy/concerntagger/`.

[6] L. Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, 2000.

[7] H. Feild, D. Binkley, and D. Lawrie. An empirical comparison of techniques for extracting concept abbreviations from identifiers. In *Proceedings of IASTED International Conference on Software Engineering and Applications (SEA'06)*, Nov. 2006.

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

[9] B. Florat. Jajuk: Advanced jukebox. Online, 2006. `http://jajuk.sourceforge.net`.

[10] E. Hill, Z. P. Fry, H. Boyd, G. Sridhara, Y. Novikova, L. Pollock, and K. Vijay-Shanker. AMAP: Automatically mining abbreviation expansions in programs to enhance software maintenance tools. In *MSR '08: Proceedings of the Fourth International Workshop on Mining Software Repositories*, Washington, DC, USA, 2008. IEEE Computer Society.

[11] E. Hill, L. Pollock, and K. Vijay-Shanker. Exploring the neighborhood with Dora to expedite software maintenance. In *ASE '07: Proceedings of the 22nd IEEE International Conference on Automated Software Engineering (ASE'07)*, Washington, DC, USA, November 2007. IEEE Computer Society.

[12] L. S. Larkey, P. Ogilvie, M. A. Price, and B. Tamilio. Acrophile: an automated acronym extractor and server. In *DL '00: Proceedings of the fifth ACM conference on Digital libraries*, pages 205–214, New York, NY, USA, 2000. ACM.

[13] D. Lawrie, H. Feild, and D. Binkley. Extracting meaning from abbreviated identifiers. In *SCAM '07: Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, pages 213–222, Washington, DC, USA, 2007. IEEE Computer Society.

[14] A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 125–135, 2003.

[15] A. Marcus and V. Rajlich. Identifications of concepts, features, and concerns in source code. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, page 718, Washington, DC, USA, 2005. IEEE Computer Society.

[16] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic. An information retrieval approach to concept location in source code. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04)*, pages 214–223, 2004.

[17] Merriam-Webster. Merriam-webster online dictionary. Online, 2008. `http://www.merriam-webster.com/dictionary/`.

[18] S. Pakhomov. Semi-supervised maximum entropy based approach to acronym and abbreviation normalization in medical texts. In *ACL '02: Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, pages 160–167, Morristown, NJ, USA, 2001. Association for Computational Linguistics.

[19] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of duplicate defect reports using natural language processing. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 499–510, Washington, DC, USA, 2007. IEEE Computer Society.

[20] D. Shepherd, L. Pollock, and K. Vijay-Shanker. Case study: supplementing program analysis with natural language analysis to improve a reverse engineering task. In *PASTE '07: Proceedings of the 7th ACM SIGPLAN-SIGSOFT*

workshop on Program analysis for software tools and engineering, pages 49–54, New York, NY, USA, 2007. ACM.

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