PARSING JAVA METHOD NAMES FOR IMPROVED SOFTWARE ANALYSIS

by

Sana Malik

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 Science with Distinction.

Spring 2011

PARSING JAVA METHOD NAMES FOR IMPROVED SOFTWARE ANALYSIS

by

Sana Malik

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 Science

Approved: _____

Pak-Wing Fok, Ph.D.
Committee member from the Board of Senior Thesis Readers

Approved: _____

Donald Sparks, Ph.D.
Chair of the University Committee on Student and Faculty Honors

ACKNOWLEDGMENTS

I would like to thank my advisors, Dr. Vijay Shanker and Dr. Lori Pollock, for their continuous support, advisement, and encouragement over the past two years. Their guidance has taught me indispensable lessons about research that I will carry for the rest of my career.

I am also grateful to Dr. Emily Hill for her mentorship from the very beginning of my undergraduate research experience. I would also like to thank the other members of the Software Analysis and Compilation lab for creating a supportive atmosphere and always being willing to help in any way they could. It has been a pleasure working with all of you.

Lastly, I would like to thank my friends and family for their support and encouragement during my undergraduate career.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Modern software engineering tools are driven by sophisticated automatic software analysis. Previous research indicates that the natural language (user-defined names) provides strong lexical clues about program behavior and structure and can be used to increase the effectiveness of various software engineering tools. Automatic analysis of the natural language usage in software requires accurate automatic parsing of the multi-word names (e.g., isPointInImage). This research focuses on developing the analysis techniques for an accurate parser for multi-word Java method names; this includes both a part-of-speech tagger and phrase chunker. These contributions form the foundation for natural language program analysis.

Chapter 1

INTRODUCTION

With program code growing larger than ever, understanding code becomes more and more difficult and time consuming. In software maintenance, developers may spend more time locating bugs and relevant code than actually fixing them. As developers read code, they look at not only the program structure, but also the names (identifiers) that the programmers have used for methods, fields, and classes in order to understand what the code does. These identifiers form the natural language of the program code. Figure 1.1 shows an example of the natural language identifiers found in program code.

Increasingly descriptive naming conventions allow the developers to capture the functionality of a method or field more precisely than before. Previous research indicates that the natural language in program identifiers provides strong lexical clues about the intended program behavior and can be used to increase the effectiveness of various software engineering tools [7], such as program search and automatic comment generation. Many such tools have been developed, but these

```
function calculateHypotenuse(legA, legB):
    hypotenuse = squareRoot(legA^2 + legB^2);
    return hypotenuse;
end
```

Figure 1.1: An example of the natural language identifiers (highlighted text) found in program code.

tools use a "bag of words" approach. That is, the relationship between words and their meanings is not known or taken into account.

In order to use the underlying lexical clues found in program code for program analysis, we must first design a tool to automatically parse the identifiers into their lexical components. Although tools exist to parse natural languages, there are none that exist specifically for the identifiers found in programming code. Non-dictionary words are commonly used because programmers often invent new words or grammatical structures when naming functions (methods) or variables. A new adjective may be formed by adding "able" to a verb (e.g., "give" becomes "givable") or the action of a method may be inferred instead of explicitly stated (e.g., a function that computes a square root may simply be named "squareRoot" instead of "computeSquareRoot"). Problems also arise when words do not follow any usual pattern (as in acronyms or domain-specific words). Because of these unique constructions, traditional parsers for natural language fail to accurately capture the lexical structure of program identifiers.

In this thesis, we

- examine the lexical and grammatical structures of Java method names,
- present techniques to parse these identifiers into meaningful lexical components, and
- preform an evaluation of the accuracy of the parser.

1.1 Related Work

Høst and Østvald [11] introduced the Java Programmer's Phrase Book, which describes commonly used grammatical structures in Java identifiers. However, the phrase book provides grammatical structures of specific words and their roles used in identifiers, whereas our system generalizes the parts-of-speech for each grammatical construction. Further, the Phrase Book only describes the phrase constructions,

whereas we also implement the grammatical rules, which includes presenting an order of application. Additionally, the constructions in the phrase book are only applied from left to right in the identifiers; our system also works right to left (which is useful when examining phrases ending with past participles).

We are interested in the applications of parsing natural language in program code for improved software engineering tools. Emily Hill [2] presents a model that uses natural language in program identifiers along with program structure to improve program search and exploration. The Software Word Usage Model (SWUM) captures conceptual information about a program through its natural language identifiers and program structure. By developing more accurate parsing and thus improving the way the natural language usage in program code is modeled, we can improve the overall accuracy of the program search and navigation provided by SWUM.

Aside from SWUM, no tool exists to automatically capture complete lexical concepts from source code identifiers. Other work in modeling the lexical structure of identifiers includes the Action-Oriented Identifier Graph(AOIG) [9]. However, the AOIG is limited to modeling verbs with their direct objects, and excludes important information such as indirect objects. Automatic tools have been developed [5, 6] to group frequently co-occuring word pairs, but these tools do not find any relationship between the words. Latent semantic analysis (LSA) [4] uses co-occurences of words to group semantically related words, but this grouping corresponds only to statistical relations, not to any linguistic relation.

Chapter 2

OVERVIEW OF PARSING METHODOLOGY

We developed a strategy for parsing Java method names by data mining and analysis on 22 open source Java programs. We extracted all the method signatures from all 22 programs. Because we are interested in tagging each word in the method name with its part of speech and program identifiers cannot contain spaces, we first had to identify the individual words in each method identifier. To do so, we used Samurai [1] an automatic identifier splitter. Samurai is able to accurately tokenize camel-case capitalized, underscore delimited, or same-case identifiers.

Our parser takes the split identifier along with the complete signature (with parameters and return type) as input, and outputs (1) the split method name with all possible tags for each word, and (2) the chunked lexical phrases representing the method name.

2.1 The Research Process

Our parsing technique is divided into two main phases: (1) tagging all possible part-of-speech of each word in an identifier, and (2) using context to choose a particular part-of-speech and chunk the identifier into its lexical components, as show in Figure 2.1. The process for developing both stages was similar and involves an iterative process of the steps:

1. Identify common word and grammar patterns ("categories")

2. Develop a generalized set of parsing rules based on word and grammar patterns

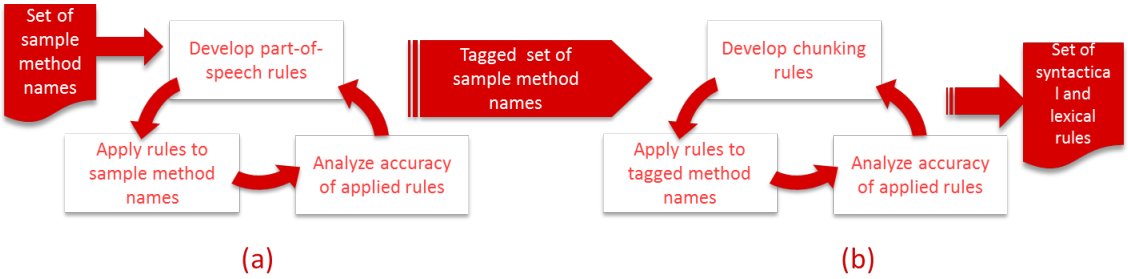

Figure 2.1: Overview of the research stages and methodology, (a) Part-of-Speech Tagging, when all possible parts of speech for the words in a method name are found, and (b) Phrase Chunking, when lexical phrases are chunked together.

3. Evaluate the effectiveness of these rules on a limited set of identifiers and

4. Expand the set of parsing rules to work on identifiers in more "categories."

With each iteration, the parser becomes more accurate. Consider the example in Table 2.1 for the method name `decodeRequest`. After the first iteration, we consider only one rule: if a word can have an "s" at the end, it is considered to be a noun (as it can be pluralized). By this rule, we mark both "decode" and "request" as nouns. We then iterate again, and add a rule for identifying base verbs: if "-ing" can be added and "-ed," then a word is likely to be a verb. Now, both "request" and "decode" are also tagged as base verbs, which is correct. However, "decode" is still tagged incorrectly as a noun. We then realize that "de-" is a prefix that is typically added to verbs, so we can eliminate this over-tagging by refining our rules further.

2.1.1 The Parsing Steps

2.1.2 Part-of-Speech Tagging

In the first step, all possible parts of speech for each word in a split method name are identified.

Iteration	Added Part-of-Speech Rule	Part-of-Speech Tags Identified
1	If `word`+s exists, `word` is a noun	decode (noun) request (noun)
2	If `word`+ing and `word`+ed exist, `word` is a verb	decode (noun, baseV) request (noun, baseV)
3	If `word` begins with "de-," `word` is only a verb, not a noun	decode (baseV) request (noun, baseV)

Table 2.1: Example of iterative refinement process for part-of-speech rules on the method name `decodeRequest`.

Morphology rules were used to determine possible parts of speech because programmers commonly use non-dictionary words when naming identifiers. Nouns and verbs are often modified by programmers to create new words that may not previously exist, but programmers follow the common patterns of the English language, and most of these new words are derived from alternate forms. For example, "poolable" is not a dictionary word, but its meaning is obvious because it follows the English grammar of adding "-able" to the verb "to pool" to form an adjective. To test if the base word (the word with its affix removed) is a valid dictionary word, we used the English dictionary from the Unix spell-checking program, Ispell.

We could not use context to provide hints about the part of speech of words because method names often omit non-critical words, and therefore do not follow the same rules as the English grammar. For example, in English sentences, a strong indicator of a noun would be those phrases preceded by a determinant ("the," "an," or "a"). However, such indicators are almost always eliminated from method names, as in the method name `createTestTable`, which omits the word "the" before "table."

Furthermore, many domain-specific words are used differently in program code than in regular English language. For example, the word "fire" is used more commonly as a noun in English, but more often as a verb in program code. For this reason, we could not use pre-existing part-of-speech dictionaries and parsers to find the parts of speech of words.

The rules that were developed for finding parts-of-speech are discussed in

Chapter 3. After all possible parts-of-speech for a word are found, the phrase is chunked into its lexical components, where a specific part-of-speech for each word is selected.

2.1.3 Chunking Phrases

After identifying all parts of speech (POS) for a given word, we can use context to select the correct part of speech in that particular instance and then group words into meaningful lexical components. Using the part of speech of surrounding words, as well as their position in a method name, we have a better idea of which POS should be selected. For instance, in the phrase "get reverse direction," reverse can be either a verb or an adjective. However, since the phrase starts with the verb "get," we can eliminate the possibility of "reverse" being a verb since it is unlikely for the phrase to have two consecutive verbs. Thus, we determine that "get" is a verb and "reverse (adj) direction (noun)" is a noun phrase for the parsing: [get]:VG [[reverse]:NM direction]:NP.

Similarly, there are various types of constructs that are commonly used in method names. For example, method names starting with "is" typically check if something is true (e.g., `isFlagSet` checks if a flag has been set). By parsing method names into phrases, we are able to extract useful information about the behavior of the method and how it interacts with relevant objects in the program.

Chapter 3

IDENTIFYING PART-OF-SPEECH TAGS

The parser begins by assigning all possible parts of speech (POS) for each word in an already split method name. The tagger takes the method signature with return and argument types, and the split method name. It outputs each word in the method name followed by a list of possible tags. All part-of-speech tags are two to five letter strings beginning with a lowercase letter. Sample input and output for a method name are shown in Table 3.1.

The part-of-speech tagger assigns tags from a set of fourteen tags, which can be divided into four categories (the tag is shown in parentheses):

- Nouns
  - Singular nouns (noun)
  - Plural nouns (pl.n)
- Verbs
  - Base verbs (baseV) and irregular verbs (irrV)
  - Third-person singular verbs (3PS)
  - Verbs ending in "-ing" (ingV)
  - Past tense verbs (pastV)
  - Past participles (pp)
- Modifiers
  - Adverbs (adv)
  - Adjectives (adj)
- Closed lists
  - Articles (art), quantifiers (quant), and pronouns (pro)
  - Prepositions (prep)

Input	`int parseCharArray(char[], int, int) | parse char array`
Output	parse (baseV) char (noun, baseV) array (noun, baseV)

Table 3.1: Sample input and output for part-of-speech tagger. Each word is followed by a list of possible tags.

Each word in a method name is considered independently from other words in the method name, so context is not taken into account at all in this phase. When assigning tags to a word, we consider all possible tags, with the exception of those in the "Closed List" category, as discussed in Section 3.5. Therefore, the order of application for the rules is not taken into account since every part-of-speech will be assigned if it is a possibility.

This chapter is divided into sections for assigning each tag category and describes the development of rules for each possible tag.

3.1 Affixing and Removing Suffixes

The tagger makes use of common prefixes and suffixes for particular word types. We call a word with no affixes a "base" word. In adding and removing these affixes to a base word, the spelling of the base word may change. In English, there are three main changes to a base word when a suffix is added:

1. Changing the last "y" to "i." This occurs when we add a suffix beginning with a consonant to a word ending with a consonant and "-y." For example, adding "-ly" to "happy" becomes "happily."

2. Remove the last "e." This occurs when we add a suffix beginning with a vowel to a word ending with a consonant, and "-e." For example, the "-ing" form of the verb "parse" becomes "parsing."

3. Double the last letter of a word. This occurs primarily when adding a suffix beginning with a vowel for words ending in "-t," "-l," "-b," "-n," "-d," "-g," as in "running," "bidding," or "jogging."

Similarly, when a suffix is removed, we must undo any spelling changes that may have occurred. Since the actual part of speech does not matter when appending suffixes, the procedure for finding base words is the same throughout all algorithms. So, without loss of generality we can create two functions that will be used throughout the tagging algorithms:

1. `append(word, suffix)` - Takes a word and suffix as input, and returns a new word with the suffix appended with the necessary spelling changes.

2. `remove(word, suffix)` - Takes a word and suffix as input, and returns the base word with correct spelling changes.

3.2 Nouns

Nouns are perhaps the most challenging type of word to identify, because their constructions are the most varied. In natural language, a strong indicator for a noun is that it follows a determinant (e.g., "the," "a," and "an"). However, as discussed in Chapter 2, these identifiers are often omitted from program identifiers.

3.2.1 Singular Nouns

The most consistent morphological identifier for singular nouns is that adding "-s" (or "-es") pluralizes a noun. However, this rule causes over-tagging (a word is tagged as a noun when it should not be) because verbs also use an "-s" ending for conjugating third-person singular. For example, the word "create" would be tagged as a noun because "creates" exists as a word. To eliminate some of this over-tagging, we remove the possibility of a word being a noun if it has a common verb prefix or suffix: "re-," "en-," "-ify," or "-ize." Further, we can eliminate over-tagging by checking if a noun suffix can be added to the word. For example, "-ance" and "-ence" are commonly added to verbs to form nouns, so such words cannot be nouns already.

Common endings for nouns are also considered: "-ity," "-tion," "-ist," "-ism," "-ness," and "-or." If a word has any of these endings, we assign the `noun` tag. For common abbreviations found in program code such as "init" and "ID", we built and check a predefined list of abbreviations.

Finally, the algorithm for deciding if a word is a singular noun is given in Algorithm 1.

Algorithm 1: `isNoun(word)`: decides if `word` is a singular noun.

```
Input: word
Output: true if word is a noun, false otherwise
1  begin
2      if word is in abbreviation list then
3          return true;
4      else if word begins with "re" or "de" or ends with "ify" or "ize"
       then
5          return false;
6      if isWord(append(word, "ance") or isWord(append(word, "ence"))
       then
7          return false;
8      if word ends with "ity" or "tion" or "ist" or "ism" or "ness" or
       "or" then
9          return true;
10     if word ends with "s" or "ch" or "x" or "z" or "sh" and
       isWord(append(word, "es")) then
11         return true;
12     if isWord(append(word, "s")) then
13         return true;
14     else
15         return false;
```

3.2.2 Plural Nouns

Plural nouns are considered primarily using the rule that plural nouns end in "-s." If a word ends in "-es" and the base word ends in an "i," we change the

"i" to a "y" and check if the base word is a noun. For "-es" words, we also check if the base word is a noun and ends with "-s," "-ch," "-x," "-sh," or "-sh." If either of these are true, then the word is tagged as a plural noun.

If the word ends in only an "-s" but does not end in "-ss," "-ius," "-ous," or "-is," we also consider it a plural noun.

The algorithm for deciding if a word is a plural noun is given in Algorithm 2.

Algorithm 2: `isPlNoun(word)`: decides if `word` is a plural noun.

```
Input: word
Output: true if word is a plural noun, false otherwise
1 begin
2     if word ends with "es" then
3         return isNoun(remove(word,"es"));
4     if word ends with "s" but not "ss" or "is" or "us" or
      isNoun(remove(word,"s")) then
5         return true;
6     else
7         return false;
```

3.3 Verbs

3.3.1 Base Verbs

Words with common verb suffixes and prefixes are tagged as base verbs (e.g., "de-," "re-," "-ify," and "-ize").

If the word can then be conjugated as a third-person singular (adding "-s" or "-es") and either of the following is true, we tag it as a base verb:

1. Adding "-ing" to the word creates a word.

2. Adding "-ed" (past tense) creates a word.

For example, "create" would be a base verb because "creates" is a word and "creating" is a word. However, "remote" would not be labelled as a base verb

because while "remotes" is a word, "remoting" is not. Out of a sample of 634 verbs, this combination of rules correctly identified 93% of verbs correctly.

We developed an irregular verb list that includes commonly used verbs in program code that are either conjugated abnormally in the English language, or are commonly used abbreviations in program code such as: "is," "has," "get," "do," "enum," and "init." These base verbs are tagged separately with `irrV`.

The algorithm for deciding if a word is a base verb is given in Algorithm 3.

Algorithm 3: `isBaseV(word)`: decides if `word` is a base verb.

```
Input: word
Output: true if word is a base verb, false otherwise
1  begin
2      if word is in irregular verb list then
3          return true;
4      if word has verb prefix or suffix then
5          return true;
6      if isWord(append(word, "s")) or isWord(append(word, "es")) then
7          if isWord(append(word, "ing")) or isWord(append(word, "ed"))
           then
8              return true;
9      else
10         return false;
```

3.3.2 Third-Person Singular Verbs (3PS)

The algorithm for deciding if a word is a third-person singular verb is given in Algorithm 4. A word is tagged as 3PS if it ends in "-s," but not "-ous," "-ius," or "-is" (since these are adjective endings) and the base word with "-s" or "-es" removed is a verb. For example, "creates" would be tagged as third-person singular because "create" is a base verb.

Algorithm 4: `is3PS(word)`: decides if `word` is a third-person singular verb.

```
Input: word
Output: true if word is a third-person singular verb, false otherwise
1 begin
2     if word ends with "s" but not "is" or "us" or "ss" then
3         return isBaseVerb(remove(word,"s")) or
              isBaseVerb(remove(word,"es"));
4     else
5         return false;
```

3.3.3 Verbs Ending with "-ing"

The algorithm for determining "-ing" verbs is given in Algorithm 5. These are verbs in their continuous form, such as "walking" and "bending." A word is tagged as an "-ing" verb if it ends with "-ing" and the base word is a verb.

Algorithm 5: `isIngV(word)`: decides if `word` is an "-ing" verb.

```
Input: word
Output: true if word is an ing verb, false otherwise
1 begin
2     if word ends with "ing" then
3         return isBaseVerb(remove(word,"ing"));
4     else
5         return false;
```

3.3.4 Past Tense Verb

The algorithm for determining if a word is a past tense verb is given in Algorithm 6. If a word ends with "-ed" and the base word is a verb, we consider it to be a past tense verb.

Algorithm 6: `isPastV(word)`: decides if `word` is a past tense verb.

```
Input: word
Output: true if word is a past tense verb, false otherwise
1 begin
2     if word ends with "ed" then
3         return isBaseVerb(remove(word,"ed"))
4     else
5         return false;
```

3.3.5 Past Participles

Past participles may function as the verb or a modifier in the English language. In program code, past participles are used in event handlers or action listeners, and typically come at the end of the method name, as in "onKeyPressed."

There are two main suffixes associated with past participles: "-en" (as in "eaten") and "-ed" (as in "brushed"), the latter of which is also used for past tense verbs. If a word ends in either of these suffixes, with a verb base word, we mark it as a past participle.

We also developed a list of irregular past participles, such as "slept" and "awoke," and any words on this list are marked as past participles.

The algorithm for determining past participles is given in Algorithm 7.

Algorithm 7: `isPP(word)`: decides if `word` is a past participle.

```
Input: word
Output: true if word is a past participle, false otherwise
1 begin
2     if word ends with "ed" or "en" then
3         return isBaseVerb(remove(word,"en")) or
          isBaseVerb(remove(word,"ed"));
4     else if word is in past participle list then
5         return true;
6     else
7         return false;
```

3.4 Modifiers

3.4.1 Adjectives

Words ending in "-able" or "-ible" with a verb base are tagged as adjective. Another good indicator of adjectives is if it is a comparative. We test this by determining if the word ends is a superlative ending in "-er" or "-est" and the base is a dictionary word. Similarly, if either "-er" or "-est" can be added to a word to form a new word, we mark it as an adjective.

We then attempt to add suffixes that require adjective bases. In particular, we use the adverb suffix "-ly" and the noun suffix "-ness." Out of a sample of 225 adjectives, these two rules alone identified 75% adjectives correctly.

The algorithm for deciding if a word is an adjective is given in Algorithm **??**.

Algorithm 8: `isAdj(word)`: decides if `word` is an adjective.

Input: word
Output: true if word is a adjective, false otherwise

```
1 begin
2     if word ends with "able" or "ible" then
3         return true;
4     if word ends with "er" or "est" and isWord(remove(word,"er")) or
      isWord(remove(word,"est")) then
5         return true;
6     if isNoun(append(word,"ness")) or isAdverb(append(word,"ly"))
      then
7         return true;
8     else
9         return false;
```

3.4.2 Adverbs

The algorithm for determining if a word is an adverb is given in Algorithm 9. The single most accurate identifier of adverbs is if the word ends in "-ly" and the base

word is an adjective. Most adverbs follow this rule, and commonly used adverbs in program code that do not follow this construction are accounted for in a predefined adverb list.

Algorithm 9: `isAdv(word)`: decides if `word` is an adverb.

Input: word
Output: true if word is a adverb, false otherwise

```
1 begin
2     if word ends with "ly" and isAdjective(remove(word,"ly")) then
3         return true;
4     else
5         return false;
```

3.5 Closed Lists

Closed lists are word types where new words are not frequently added, mainly because there are no morphology constructions for these parts of speech. For example, articles are limited to "the," "a" or "an" and no new articles can be formed. Closed lists are represented as predefined word lists in the system, and words are determined to be any of these parts of speech by checking for membership in these lists.

Articles, `art` There are only three articles in the English language: "the," "an," and "a."

Pronouns, `pro` **and quantifiers,** `quant` Often, the quantifiers are used as pronouns, so we include these in a predefined list. For example, the quanitifer "both" may describe a plural noun, but may be used on its own as a pronoun, as in the phrase "get both."

```
all
another
anybody
anyone
anything
both
each
everything
he
her
it
most
much
my
nobody
none
```

others	this
our	those
ours	us
someone	whoever
theirs	whom
them	your
they	yours

Prepositions, `prep` Prepositions are words such as "to," "of," or "on."

The closed list tags differ from all tags in the system because when a word is identified as one of these parts of speech, it does not assign any other possible tag to the word.

Chapter 4

CHUNKING PHRASES

After identifying all possible parts of speech for the words in an identifier, the parser begins chunking the method name. The chunker takes the output from the tagger (each word in a method name followed by a list of possible tags) as input and outputs a bracket-separated parsing of the method name. Sample input and output for the chunker are shown in Table 4.1. Each phrase is contained inside a set of square brackets, and the right bracket is suffixed with its phrase label. Tags for individual rules are removed as they are implied by the lexical phrase in which the word is contained. It is possible for lexical phrases to be nested, as in the case of a noun modifier phrase inside a noun phrase.

Typically, method names consist of a verb group describing the *action* of the method names followed by direct or indirect objects which refer to the objects the method is acting upon, as in "create tree from constructor." In some cases, the direct or indirect objects may be inferred from the method's class or arguments, respectively. In these cases, the method name does not include the corresponding direct or indirect object. Consider the method name `moveTo(Coordinates xy)`. The method implies that we are moving something to the coordinates provided in the arguments. Most likely, we are moving an instance of the class that the method is a part of.

Similarly, the method name may infer the action by providing only the direct or indirect object. A common example is a method name such as "toString" which

Input	parse (verb) char (noun, baseV) array (noun, baseV)
Output	[parse]:VG [[char]:NM array]:NP

Table 4.1: Sample input and output for phrase chunker. "VG" denotes a verb group, "NM" denotes a noun modifier, and "NP" denotes a noun phrase.

implies that we are *converting* an object to a string. It is also common to not include "calculate" or "compute" (as in `squareRoot` for "compute square root").

The third type of common method name is that set of names that are concerned with the state of an object. These method names are typically (1) boolean functions that check if something about an object is true (e.g., *isCachable*), or (2) listener methods that respond to an action that just happened (e.g., `activityStarted`).

To represent a method's behavior through its parsed lexical components, we use five basic types of lexical phrases. All grammatical constructions are derived from combinations of one or more of these basic lexical phrases.

In this chapter, we discuss the basic lexical components of method names, the grammatical constructions that we developed and the way the system chooses the most likely construct for a given method name.

4.1 Targeted Lexical Phrases

Our system uses five of the same lexical phrases that are commonly found in the English language. In this section, we describe each type of phrase, give examples of how it is found in method names, and present the matching pattern used in the chunker.

4.1.1 Noun Modifier Phrase (NM)

Description A noun modifier phrase is any combination of nouns, adjectives, or determinants that precedes a noun. For example, in "table row header", "table row" is a noun phrase, but since it is modifying the noun "header," we mark it as a noun modifier phrase.

Pattern NM $\rightarrow (noun|adj|pp|ingV|quant|pro)^*$

4.1.2 Noun Phrase (NP)

Description Noun phrases consist of any type of noun (singular, plural, gerund, or pronoun), optionally preceded by a noun modifier phrase. An example is the method name "utc encoding limit" or "new object" in "create new object." Two noun phrases joined by "of" also make a noun phrase, as in "size of array".

Pattern NP $\rightarrow$ (NM) $noun|pl.n|ingV|pro$

4.1.3 Verb Group (VG)

Description A verb group ends with any word from the verb category (ingV, baseV, 3PS, pastV, or irrV) and is optionally preceded by helping verbs or verb modifiers (VM), such as adverbs.

Pattern VG $\rightarrow$ (VM) $baseV|ingV|3PS|pastV|irrV$

4.1.4 Past Participle Phrase (PP)

Description A past participle phrase is noun phrase followed by a past participle. For example, in the method name `isButtonPressed`, "button pressed" is a past participle phrase.

Pattern PP $\rightarrow$ *pp*
PP $\rightarrow$ NP ("is"|"has been") *pp*

4.1.5 Prepositional Phrase (prepP)

Description A prepositional phrase is a noun or past participle phrase preceded by a preposition. For example, in the method `movePanelToFront`, "to front" is a prepositional phrase.

Pattern prepP $\rightarrow$ *prep* NM
prepP $\rightarrow$ *prep* PP

4.2 Grammatical Constructions of Method Names

All chunking rules consist of a combination of one or more of the basic lexical phrases. Our system uses sixteen grammatical constructions, which we divide into three categories: verbs and objects, and other cases.

4.2.1 Verb Groups and Objects

These constructions contain explicit actions in the form of a verb group. In many cases, the method acts upon an object, which may or may not be indicated in the method name.

4.2.1.1 Verb Group (VG)

Description Method names that contain only a VG are usually one word method names.

Examples

`void delete(int)`	[delete]:VG
`int insert(Object)`	[insert]:VG
`void enable()`	[enable]:VG

Chunking The chunker checks if the phrase ends with a verb, and if so checks if the rest of the method name is a verb modifier phrase.

4.2.1.2 Verb Group and Noun Phrase (VG NP and NP VG)

Description A verb group followed by a noun phrase represents a method name with an action and direct object. It is the most common type of method name. Less commonly, the method name can be structured like a typical English sentence, with the subject (NP) first and the verb group second.

Examples

`void checkCriticalTasks(Task,List)`	[check]:VG [[critical]:NM tasks]:NP
`Choice showDialog(Component)`	[show]:VG [dialog]:NP
`boolean experimentIndexExists()`	[[experiment] index]:NP [exists]:VG

Chunking The algorithm splits the phrase after the first verb, and tests if the first part of the phrase forms a verb group and if the second part forms a noun phrase, or vice versa for the NP VG case.

4.2.1.3 Verb Group, Noun Phrase, and Prepositional Phrase (VG NP prepP)

Description A verb group followed by a noun phrase and prepositional phrase represents a method name with an action and both direct and indirect objects.

Examples

`File findFileInPaths()`	[find]:VG [file]:NP [in [paths]:NP]:prepP
`void DoPasteFromClipboard()`	[do]:VG [paste]:NP [from [clipboard]:NP]:prepP

Chunking The algorithm splits the phrase after the first verb, and tests if the first part of the phrase forms a verb group and if the second part forms a noun phrase with a prepositional phrase.

4.2.1.4 Verb Group and Prepositional Phrase (VG prepP)

Description A verb group followed by a prepositional phrase represents a method name with an action and an indirect object. Usually they infer a direct object (usually the class to which they belong). These method names are not as common as other verb and object combinations.

Examples

`void goToChild(int)`	[go]:VG [to [child]:NP]:prepP
`void goToNextStep()`	[go]:VG [to [[next]:NM step]:NP]:prepP

Chunking The algorithm splits the phrase after the first verb, and tests if the first part of the phrase forms a verb group and if the second part forms a prepositional phrase.

4.2.1.5 Verb Group, Noun Phrase, and Preposition (VG NP prep)

Description A verb group followed by a noun phrase and preposition represents a method name with an action and a direct object, with the indirect object inferred by the object the method is called on.

Examples

`Object extractElementAt()`	[extract]:VG [element]:NP at
`void DoSendTo()`	[do]:VG [send]:NP to
`void insertNodeAt(Node,int)`	[insert]:VG [node]:NP at

Chunking The algorithm splits the phrase after the first verb, and tests if the first part of the phrase forms a verb group and if the second part forms a noun phrase followed by a preposition.

4.2.1.6 Verb Group and Preposition (VG prep)

Description A verb group followed by a preposition represents a method name with an action and the indirect and direct objects are both inferred by the class and method arguments.

Examples

`int compareTo(Object)`	[compare]:VG to
`void dragOver(DropTargetDragEvent)`	[drag]:VG over

Chunking The algorithm checks if the last word is a preposition and if the rest of the phrase is a verb group.

4.2.2 Other Cases

Sometimes a method may not have an action associated with it. In these cases, the action is either implied or has already happened (as in the case of past participles). Because the action is implied, the name of the method will describe what the function returns.

4.2.2.1 Noun Phrase (NP)

Description Method names that contain only a NP are typically constructors, though they also occur for commonly inferred actions such as "get" or "compute." Constructors are discussed in Section 4.2.2.5.

Examples

`void anotherConfig`	[[another]:NM config]:NP
`int squareRoot(int)`	[[square] root]:NP
`Coords lineIntersect()`	[[line]:NM intersect]:NP

Chunking Test if the last word is a noun, and if so, check if the rest of the phrase is a noun modifier.

4.2.2.2 Past Participle Phrase (PP)

Description When a past participle phrase occurs by itself in a method name, it typically describes an action that has happened or the state of an object. Consider the example method `heapifyExtended`. From left to right, we would assume "heapify" to be the verb, which would leave "extended" to be treated as a modifier. However, if we were to describe this method in natural language, we would say that the action of "heapifying" has been "extended." With this, it is clear that "extending" is the action that is being applied to "heapifying."

Further, ambiguity is caused because method names do not contain the verb helpers that may be found in natural language. For example, in the method `keyPressed` there are two possible interpretations:

1. "pressed" describes the state that the key is in (i.e., key *is* pressed, and "pressed" is modifying "key").
2. "pressed" is an action that has occured (i.e., key *has been* pressed).

In natural language, the connotation would be clear. However, in the interest of shortening identifiers in program code, these helpful contextual words are left out. Instead, we have found a good indicator of this is the return type of the method. In methods where the past participle describes a state, the return type is often a boolean. In method names where the participle represents an action, the return type is void.

Examples

`void actionPreformed(ActionEvent)`	[[action]:NP preformed]:PP
`void mouseReleased()`	[[mouse]:NP released]:PP

Chunking Check the last word for a past participle first, then the rest of the phrase for a noun phrase.

4.2.2.3 Adjectival Phrase (adjP)

Description When a noun modifier appears in a method name alone, we consider it to be an adjectival phrase that refers to an object preceding the adjective, or an inferred object if no noun phrase is present. For example, the method `Object previous()` finds the previous object in a list, and the method `boolean anyEarlier(AuctionEntry)` checks if there are any auction entries earlier than the one given.

Examples

`Object previous()`	[previous]:adjP
`boolean anyEarlier(AuctionEntry)`	[any earlier]:adjP

Chunking If the method ends in an adjective, find the first noun from right to left to form a noun phrase (if applicable). The rest is the adjectival phrase.

4.2.2.4 Prepositional Phrase (prepP)

Description Within a method name, prepositional phrases represent indirect objects. Methods that contain only a prepositional phrase often infer some action like "convert," "get," or "move."

Examples

`byte[] toWindowsName(String)`	[to [[windows]:NM name]:NP]:prepP
`XMLElements toXML()`	[to [XML]:NP]:prepP
`void onOkPressed()`	[on [[ok]:NP pressed]:PP]:prepP

Chunking If the identifier begins with a preposition, we check the rest of the name for a phrase or past participle phrase.

4.2.2.5 Special Cases

Special cases are method names that appear commonly and follow a typical naming convention, such as constructors, accessors ("get" followed by a field name), and mutators ("set" followed by a field name). Because of this, these method names do not need to test for applicability of the other grammatical constructions. Instead, we test method names as one of these common types and automatically apply the correct chunking.

4.2.2.5.1 Constructors

Description Constructors are always class names, and classes typically represent objects in a program. Because of this, if a method name is a constructor, we automatically treat the entire phrase as a noun phrase.

Examples

`MoveToFrontAction(DrawingEditor)`	[[move to front]:NM action]:NP
`WarehouseGoodsPanel(Colony)`	[[warehouse goods]:NM panel]:NP
`OutDegreeFunction()`	[[out degree]:NM function]:NP

Chunking The last word of the constructor is treated as the head of the noun phrase, and the rest is chunked as a noun modifier.

4.2.2.5.2 "Is," "Can," and "Has"

Description We treat methods that begin with "is," "can," or "has" together because these constructs are commonly followed by a boolean field or an object name. These constructs are classified as verbs followed by noun modifiers, since boolean field names describe a state, as indicated in the previous section. Similar constructs that do not begin with these three verbs are considered as VG NM or VG PP cases.

Alternatively, non-boolean field names or constructors typically still represent an object. Because of this, constructors and phrases following "get" and "set" are automatically parsed as noun phrases. In a sample of about 400 method names containing multiword field names, 78% began with "get" or "set."

Examples

`BibTexEntry getSingleCitation(String)`	[get]:VG [[single] citation]:NP
`boolean isAimingAtLocation()`	[is aiming]:VG [at [location]:NP]:prepP
`boolean isNetworkEnabled`	[is]:VG [[network]:NP enabled]:PP

Chunking After extracting the verb group, the rest of the method name is treated as a past participle phrase, noun phrase, or adjectival phrase.

4.3 Ordering the Rules

Because words typically have multiple parts of speech, it is possible to apply more than one grammar construction to any method name. For example, the method name *registerID* can be parsed as either [register]:VG [id]:NP or [[register]:NM id]:NP meaning it could refer to registering an ID, or a register ID. Since we want to select a single parsing for each method name, we need to determine which parsing is most likely for ambiguous method names.

Because the cases described in Section 4.2.2.5 override the regular grammar constructions, we attempt to apply those rules first. If no applicable construction is found, we move on to the lexical phrases and combinations.

In preliminary research, we took all possible applications of constructs, then evaluated manually which constructs should be applied. There were primarily four combinations of overlapping constructs. Within each group, an application order was determined based on which should be selected most often.

- VG PP and PP - In method names such as "fire activity started" and "close button pressed," it is difficult to determine whether the first verb is the action, or if it is part of the action described by the past participle. Usually, the verb describes the response to the action, so VG PP is selected over PP.

- VG NP, NP VG and NP - We found that out of a sample of 50 method names, 30 could be of NP or VG+NP. Of these 30 ambiguous method names, manual analysis showed that 24 should be parsed as VG+NP instead of as NP.

- NP or VG - Single word method names such as "write" or "size" can be parsed as either a noun phrase or verb group. Manual analysis showed that one-word method names are more likely to be verbs, though both types are common.

- prepP and VG prepP - On analysis of 10 ambiguous method names, VG prepP was most commonly the rule that should be selected.

Based on this analysis, more common rules are applied before less common ones. A figure illustrating the final order of application is shown in Figure 4.1.

In ordering the grammatical rules this way, we are able to chunk most method names accurately. However, there are still certain method names that are parsed incorrectly because of ambiguity. For example, the method name `lineIntersect` would be parsed as [line]:VG [intersect]:NP when it should be parsed as [line intersect]:NP, since the `VG NP` construct is more likely than `NP`.

4.4 Context Frequencies

Until this point, we only look at how the word appears in the context of the given method name. However, in the case of exceptions, this is not a good indicator of how these words most commonly appear. By looking at how a word appears in other method names, we can determine what role the word most frequently appears in. In particular, this is helpful in deciding whether a word acts an a verb versus a noun, as in the "line intersect" example above in Section 4.3.

We define common "verb" positions to be the beginning of method names and singular method names. Common "noun" positions are the end of methods and fields and singular field names. Using these rules, we took the 200 most commonly occurring words in program code, and counted how many times they appear in each of these positions. Ordered by the ratio of verb positions to noun positions, we determined thresholds for when a word is certainly a verb and when a word is certainly a noun. Before a method is parsed, we test for any possible ambiguity. If a phrase can be parsed as both a NP and NP VG, we attempt to narrow the tags to how it most frequently occurs. If the frequency ratio does not provide strong evidence for either a noun or a verb, we continue parsing with both possibilities.

Word	Verb Occurrences	Noun Occurrences	Ratio (V/N)	% Verb
Can	575	35	16.4	94.3%
Set	16102	1090	14.8	93.9%
Handle	1169	115	10.2	91.0%
Fire	331	60	5.5	84.7%
Test	1890	393	4.8	82.8%
Map	63	908	0.069	6.5%
Function	21	329	0.063	6.0%
View	46	754	0.061	5.7%
Count	77	1426	0.053	5.1%
Name	61	3881	0.015	1.5%

Table 4.2: Results from mining context frequencies of common words found in program identifiers. The words can be either verbs or nouns, and context frequency helps determine which is more likely. A high ratio indicates a verb is more likely, while a low ratio indicates that a noun is more likely.

Otherwise, we narrow the tags to whichever POS is more likely, and then continue parsing. Table 4.2 shows examples of verb-noun ambiguous words and resulting tags.

4.5 Bringing It All Together

After all the rules are developed and the order is determined, they are applied in succession until the method name matches one of them correctly. Context frequency analysis is done after applying special cases, then the rest of the rules are applied in the order described in Section 4.3. Figure 4.1 illustrates the chunking process and order of rule application.

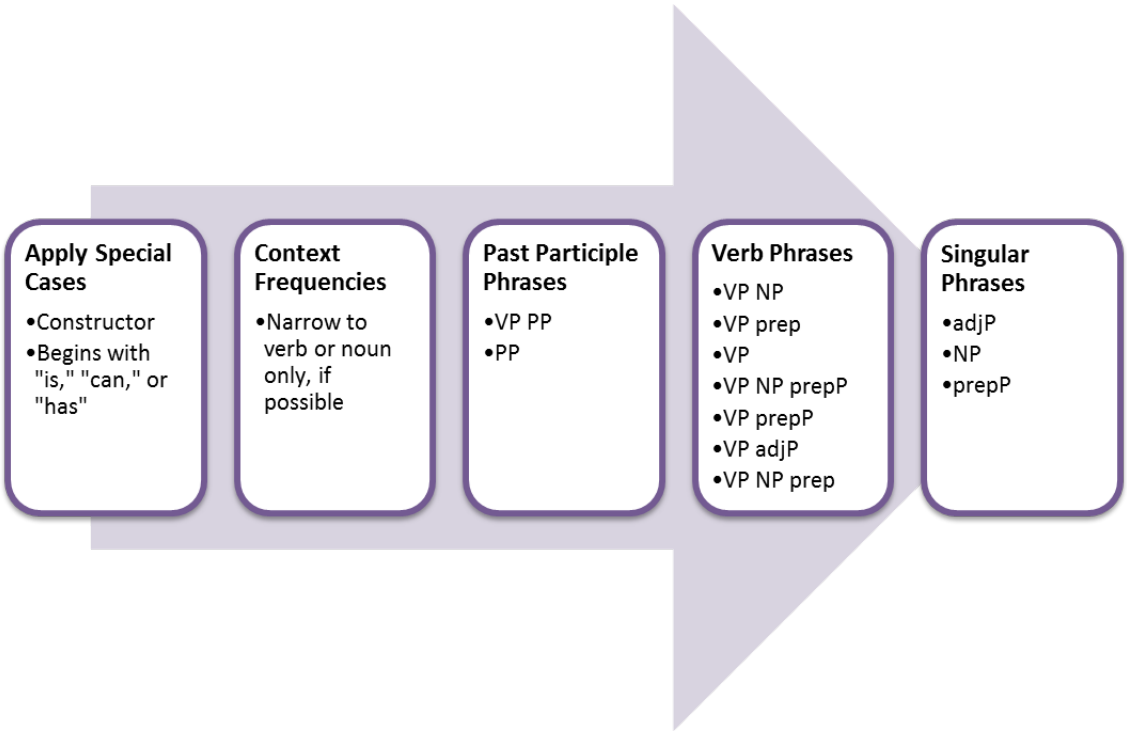

Figure 4.1: Order of grammar rule application.

Chapter 5

EVALUATION

In this chapter, we evaluate the parser for accuracy against ideal parsings that capture the lexical components for a method name. We evaluated the parser for accuracy of the overall parsings it produced. We did not look at the accuracy of the part-of-speech tagging for individual words because a specific tag is not selected until the phrase is being chunked, and after that point the part of speech is inferred from the chunking.

5.1 Methodology

We evaluated the parser on a set of 195 method names that are representative of parts-of-speech and phrase constructions from all developed categories. These method names were randomly chosen from the 22 open source Java programs used throughout the research process.

We parsed each method name manually to create a set of "ideal" parsings. Each "ideal" parsing was then compared to the output of the automatic parser. To compare, we evaluated the number of successes (where the automated parsing matches "ideal" parsing) with the number of errors. The incorrect parsings were then evaluated for specific reasons for failure.

5.2 Results

Of the 195 method names, 191 were parsed correctly and 4 were parsed incorrectly. Of these four incorrect parsings, one was caused by a tagging error, one

was caused by context frequency thresholds, one was a construction the parser does not handle, and the last was caused by rule order. All of these method names were exceptions to rules that otherwise make the system more accurate.

5.2.1 Errors

Method Name: `boolean isMultiline()`

Automatic Parsing: [is]:VG [multiline]:NP

Ideal Parsing: [is]:VG [multiline]:adjP

This error was caused by the tagger. The word "multiline" is a construction of an adjective describing multiline comments. However, the tagger does not recognize it as an adjective based on the rules we use, so it is marked as a noun (the default if no other tag is applied).

Method Name: `ImageMedia countColors(ImageMedia)`

Automatic Parsing: [[count]:NM colors]:NP

Ideal Parsing: [count]:VG [colors]:NP

An issue with the context frequency is when a word commonly occurs as a noun, but sometimes as a verb. In this case, "count" appears 1426 times as a noun, but only 77 times as a verb (about 5%), so it is marked as a noun. However, this method name represents one of the times it *does* act as a verb.

Method Name: `AST createUsingCtor(Token, String)`

Automatic Parsing: [create]:VG [[using]:NM ctor]:NP

Ideal Parsing: [create]:VG [using [ctor]:NP]:VP-ingV

The method names follows a construction which has two verb phrases and is uncommon for method names. Because of this, our parser does not account for this type, and adding this construction may affect the accuracy of other rules, so further analysis would need to be done before adding it. In particular, treating the gerund

"using" as a preposition could cause problems for "-ing" verbs that do not act as a preposition (since this is more common).

Method Name: `void closeButtonActionPerformed(ActionEvent)`
Automatic Parsing: [close]:VG [[button action]:NM preformed]:PP
Ideal Parsing: [[close button]:NM action]:NP performed]:PP
Since this method name begins with a verb and ends with a past participle, there are two possible rule applications: verb group followed by a past participle phrase or a singular PP containing a noun phrase. The correct parsing would be to parse "close button action" as a noun phrase inside the past participle phrase. However, in other rules with both of these parses, the verb group, past participle is correct more often, so the grammar applies that rule first.

Chapter 6

CONCLUSIONS AND FUTURE WORK

The natural language found in program identifiers provides strong lexical clues about program behavior and can be used to increase the effectiveness of various software engineering tools. Our system parses Java method names into their lexical phrases as a basis of understanding these lexical clues. With the parser, software engineering tools can then take advantage of these clues for improved software analysis, such as program search and navigation. Preliminary evaluation indicates that our parser is highly accurate, parsing about 97% of the 195 method names in our evaluation set correctly.

In future work, we plan to continue to improve the accuracy of this tool by integrating abbreviation expansion to take advantage of the additional semantics it would provide. Additionally, we would like to extend our research to not only method names, but also field names and generalize the system to work on identifiers in different programming languages.

BIBLIOGRAPHY

[1] Eric Enslen, Emily Hill, Lori Pollock, and K. Vijay-Shanker. Mining source code to automatically split identifiers for software analysis. In *6th IEEE International Working Conference on Mining Software Repositories*, 2009.

[2] Emily Hill. *Integrating natural language and program structure information to improve software search and exploration.* PhD thesis, University of Delaware, August 2010.

[3] Emily Hill, Lori Pollock, and Vijay Shanker. Automatically capturing source code context for software maintenance and reuse. *International Conference on Software Engineering (ICSE)*, May 2009.

[4] Thomas K. Landauer, Danielle S. McNamara, Simon Dennis, and Walter Kintsch, editors. *Handbook of Latent Semantic Analysis.* Erlbaum, Mahwah, NJ, USA, 2007.

[5] Yoelle S. Maarek, Daniel M. Berry, and Gail E. Kaiser. An information retrieval approach for automatically constructing software libraries. In *IEEE Transactions on Software Engineering*, volume 17, pages 800–813, 1991.

[6] Chris Manning and Hinrich Schutze. *Foundations of Statistical Natural Language Processing.* MIT Press, Cambridge, MA, USA, May 1999.

[7] Lori Pollock, Vijay Shanker, David Shepherd, Emily Hill, Zachary Fry, and Kishen Maloor. Introducing natural language program analysis. In *7th ACM SIGPLAN-SIGSOFT Workshop of Program Analysis for Software Tools and Engineering*, June 2007.

[8] David Shepherd, Zachary Fry, Emily Hill, Vijay Shanker, and Lori Pollock. Using natural language program analysis to locate and understand action-oriented concerns. In *International Conference on Aspect-Oriented Software Development*, April 2007.

[9] David Shepherd, Lori Pollock, and Vijay Shanker. Towards supporting on-demand virtual remodelization using program graphs. In *5th International Conference on Aspect-Oriented Software Development*, 2006.

[10] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and Vijay Shanker. Towards automatically generating summary comments for java methods. In *25th IEEE International Conference on Automated Software Engineering (ASE'10)*, September 2010.

[11] Einar W. Høst and Bjarte M. Østvold. The java programmer's phrase book. pages 322–341, 2009.