Automatic Summarization of Source Code for Novice Programmers

Wyatt Olney
Drew University, 2016

ABSTRACT

The process of generating part-of-speech information is a well established problem in the field of computer science. A wide variety of taggers exist, and have been trained to use english text, and extract this information automatically. However, these taggers are traditionally only used for parsing information from traditional written English, such as news articles. Many of these taggers are evaluated on the Wall Street Journal corpus, which consists of many such articles. However, natural language artifacts also appear in the corpus of software source code, such as in method names. This thesis proposes a methodology for comparing these taggers on source code artifacts, and evaluating their overall accuracy. Additionally, a potential application of part-of-speech tagging source code is presented in this thesis. Specifically, a tool for novice programmers is developed and shown how this could be improved using this linguistic information to generate better, and more detailed summaries for novices, by extracting information from method names. These types of summaries would allow beginning programmers to learn how to read and work with code written by others. This is a major component of learning to work with code, especially with the collaborative nature of many modern software projects. By generating summaries automatically, the daunting appearance of production level source code becomes easier to broach and understand for a novice.

Automatic Summarization of Source Code for Novice Programmers

by
Wyatt Olney

An Honors Thesis
Submitted in Partial Fulfillment of the
Requirements for the Degree of Bachelor in Arts with
Specialized Honors in Computer Science
at Drew University
May 2016

Copyright by

Wyatt Olney

2016

ACKNOWLEDGMENTS

I would like to thank Dr. Emily Hill for her guidance and assistance in her role as primary advisor to this thesis. Furthermore, I would like to thank Dr. Steven Kass for his assistance in providing detailed statistical analysis. Additionally, I would like to thank Dr. Patrick Dolan and Dr. John Muccigrosso for their feedback on writing. I also want to acknowledge the assistance of Bezalem Lemma for his assistance in the annotation of the gold sets, and Chris Thurber for his assistance in writing several scripts to help analyze the data. Finally, I would like to thank Drew University and the Department of Mathematics and Computer Science for allowing me to conduct the research presented in this paper.

Contents

Ch. 1. Introduction 1

1.1 Learning to Read Code 2

1.2 POS Tagging and Comment Generation 3

Ch. 2. Background and Related Work 8

2.1 Identifiers 9

2.2 POS Taggers 10

2.3 Comment Generation 12

2.4 Natural Language Artifacts 14

2.5 Program Tutors 16

Ch. 3. Automatically Generating Explanations of Source Code for Novice Programmers 18

3.1 Selection of Data for Analysis 19

3.2 Summary Overviews 21

3.3 Programmatically Summarizing Source Code 25

3.4 Tool Design and Requirements 27

3.4.1 Extension Points Within Eclipse 28

3.4.2 Overview of Eclipse Architecture 29

3.4.3 Using Code Teacher 31

3.5 Inclusion of POS information 34

Ch. 4. POS Tagging Identifiers in Source Code 35

4.1 Selection of Taggers 36

4.2 Experimental Gold Set 39
4.3 Experiment Design 41
4.4 Results & Analysis 47
4.5 Discussion & Qualitative analysis 53

Ch. 5. Conclusions and Future Work 59

Bibliography 62

Appendix Ch. A. CodeTeacher Plug-in Source Code 70

Appendix Ch. B. Gold Sets 73

Chapter 1

Introduction

As technology has become more prominent in everyday life, the need for individuals who are capable of working with code and developing new software has increased greatly. Due to this increased need, there has been a similar need for improved education for students in computer science. Additionally, as software has become more complex, the number of developers working together on even small projects has become larger, requiring greater collaboration between multiple developers. A major part of this cooperation is learning to work with code written by other people. This can be a difficult task to learn, especially for novice programmers, as learning this requires the programmer to understand the logic of the program, the syntax of the code, and how the different elements of the code fit together, all of which may be unfamiliar to a novice programmer.

1.1 Learning to Read Code

Source code written to be used in software can be difficult for a novice programmer to understand. Beyond simply coming to terms with the logic of how a program works, a novice programmer needs to also learn the specific syntax of a language, as well as specific function calls that are part of the standard library of the language.

Currently, a student could ask a teacher or another student what a piece of code does, or for assistance in understanding how a section of code works. A student may have the option to read any comments in the code, but if the code is not well maintained, this is not always a viable option, as comments may be out of date or incomplete. Additionally, inserting comments into code can be a time consuming process for a developer, meaning that inserting these comments into source code can be a low priority task during the writing of new code, so many projects have little documentation [37]

However, these are not the only sources of information of how a piece of code is run. Within the code itself, there exists a fair amount of extra information. This includes small pieces of linguistic information, such as variable, class, and method names. Any of this information is useful for a programmer to help figure out what a piece of code does. However, for the actual execution of the program, a computer ignores this information, because it is unnecessary to the actual execution. Figure 1.1.1 illustrates several examples of source code natural language artifacts that hint at what these variables and method names do.

```java
public String getFormatedValue(String sKey){
    String sOut = null;
    Object o = getValue(sKey);
    PropertyMetaInformation meta = getMeta(sKey);
    if (meta.getFormat() != null){
        if (meta.getType().equals(java.util.Date.class)){
            try{
                sOut = meta.getFormat().format((Date)o);
            }
            catch(Exception e){ //parsing error
                Log.error("137",e); //$NON-NLS-1$
            }
        }
    }
```

FIGURE 1.1.1: Examples of natural language artifacts in source code

1.2 POS Tagging and Comment Generation

In order to utilize information from natural language artifacts, computers need to be able to process the natural language artifacts which appear in the code. Fortunately, there exists a variety of software tools to label the parts of speech of written language, commonly known as part-of-speech (POS) taggers. The problem with these software tools is that they were developed for, and trained on, a natural language corpus, generally the Wall Street Journal corpus. An example of this kind of output can be seen in figure 1.2.1. Unfortunately, the types of language for which the taggers are built and trained are very different than the types of language which appear within source code for software [39].

Another challenge is that source code language uses short phrases, rather than full, structured sentences as can usually be seen in writings in natural language. Additionally, a further challenge for part-of-speech tagging source code is that, for

The boy walked
Determiner noun past tense verb
down the street.
Preposition determiner noun

FIGURE 1.2.1: Output of sample POS tagger on a natural language sentence [17]

get current volume
Verb Adjective Noun — Good

print stack trace
noun noun noun — Poor

FIGURE 1.2.2: Output of sample POS tagger on two method calls from source code

the convenience of the programmers working on it, abbreviations are used frequently. These make typing statements faster and more convenient for the programmer, but taggers may treat abbreviations as unknown or foreign words, a tag which provides little or no syntactic information.

Additionally, as can be seen in figure 1.2.2 the tagging of source code can be inaccurate. While in the first example, "get current volume", the tagger recognized that this was intended to be a full statement, with a verb and an object, in the second example, "print stack trace", all three words were interpreted to be nouns. While all of these words can be nouns, it would be more accurate to say that print is a verb, with trace being the main object, and stack being an adjective. For this reason, the automatic tagging in this example is weak.

The difficulties in analyzing natural language artifacts results from the ambiguity

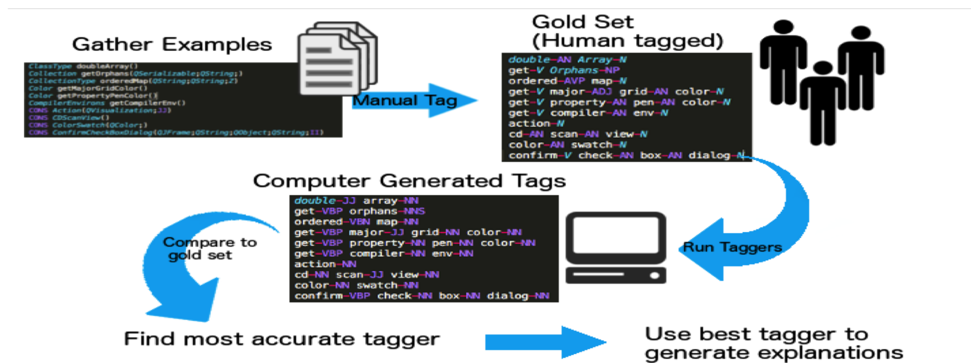

FIGURE 1.2.3: Overview of testing part-of-speech taggers

of many words in natural language. For example, the same word may be a noun or verb depending on how it is used in the context of other words in the statement, such as what can be seen in the example of "print stack trace", in figure 1.2.2, where all words could be interpreted as nouns, or as other parts of speech. This problem has largely been approached by researchers who have developed part-of-speech taggers which can incorporate data from large corpuses, and use statistical analysis to generate highly accurate tags. However, most of these taggers use the statistical model for natural language, which is grammatically different from source code artifacts. For this reason, it becomes necessary for this thesis to evaluate the effectiveness of existing POS taggers on the corpus of source code artifacts. The process of testing the part-of-speech taggers is summarized in figure 1.2.3.

The reason that the part-of-speech of words in source code needs to be found is that this information can help to determine the action that is being performed (by the verb phrase), and on what the action is performed (noun phrase). This type of understanding is something which an individual reading a piece of code will do intuitively,

given that they have some experience with reading code. With this information, along with additional information that is delivered from the source code's compilation, an effective summary could be created. The information from the compiler allows for a program to be able to parse what an individual line of code does at a low level, while parsing the source code for information from natural language artifacts gives a higher level summary of what a line of code does, i.e. what is being acted upon, and what is being done to it. This first piece of information can be recognized from the verb phrase, and the later comes from the noun phrase, if these pieces exist in the artifacts.

The novelty of this thesis comes from the fact that while all of these research programs have focused on automating the generation of comments and documentation for source code, the solution proposed in this thesis focuses on generating summaries dynamically. Additionally, the summaries that will be generated are designed to be helpful to those with little to no experience with programming. Previous work has illustrated that novices approach the problem of program comprehension differently from experts. Novice programmers spend significantly longer per line to understand how code works [10]. Additionally, while a novice will likely need to spend some of their work time on simply understanding the syntax of a line of code, an expert will be able to understand most, if not all, code syntax very quickly, and would not struggle with understanding what a piece of code does in this way. Experts generally need only high level overviews of code to understand where they should focus their time, while novice programmers may be less able to isolate these problems and understand code.

This thesis makes the following contributions:

1. The evaluation of preexisting part-of-speech taggers on the nonstandard language of source code, which differs from the natural language writings that part-of-speech taggers are usually evaluated on.

2. The development of software (specifically, an Eclipse plug-in) which can parse, label, and process the natural language artifacts found in source code to develop explanatory statements for novice programmers.

3. A proposal for how part-of-speech information could be incorporated into a summary tool similar to the one we wrote, to improve the accuracy of summaries of source code.

Chapter 2

Background and Related Work

This thesis draws upon several existing fields of research, and also makes contributions to two of these fields. The first field of research is part-of-speech tagging. Research has also been carried out in creating tools which can autonomously teach programing and coding to novice programers. These tools are similar to the program developed as part of this paper, but focus on a different aspect of learning how to code. This research has also been applied to the field of software engineering, to help even experienced programmers learn to work with new code more quickly. Much of this research is related to the automatic maintenance and documentation of code, using natural language artifacts to generate documentation and explanatory comments automatically, without needing additional time and effort from developers.

2.1 Identifiers

In order to summarize code, it is essential to know both the syntax of the language in which the code is written, and what, at a high level, the code is intended to do. The first of these two tasks will be addressed later in this thesis. However, much of the information about what code does is available to an individual with training in how to read code. Because programmers name variables, methods, and other artifacts (which are also known as *identifiers*) in source code in whatever way they want, there are no technical reasons why the chosen names must be informative. However, when Liblit, Begel, and Sweetser studied program source code identifiers [28], they found that programmers do in fact use informative, meaningful names for variables and values, particularly in languages such as java that do not limit the length of names in code. This indicates that using these pieces of information will return meaningful information. However, it is worth noting that they also found that abbreviations were common in code, since writing a long name many times quickly became counterproductive. This is potentially threatening to this thesis because these abbreviations introduce ambiguity to the tagging process, as an abbreviation may stand for several different words, and additionally may not be recognized by the tagger as English words, which would prohibit accurate POS tagging. Furthermore, in this study, it was found that the more visible an identifier is in the code, the more likely it is to be informative and expressive. This means that method names, classes, and other high-level organizational components of code are more likely to possess informative names than low-level constructs, such as local variables. This suggests that the most effective way to summarize a piece of code is to focus mostly on high

level organization, rather than using local variables, which provide less information.

2.2 POS Taggers

POS tagging is the process of taking a body of text and labeling the part-of-speech of the individual words. A large variety of tools have been developed for the purpose of analyzing this automatically [6, 41, 35, 42, 7, 18]. Many of these taggers have accuracy on their evaluated corpuses that reaches over 90%, making these taggers highly reliable on natural language. Additionally, some of these tools are developed for specific languages and purposes, such as parsing and labeling biomedical texts [43].

Source code artifacts are significantly different than natural language. Source code artifacts, while following strict syntactical rules, as defined by the language, tend to not follow strict grammatical rules, and often include abbreviations, which could interfere with the ability of POS taggers to process the language effectively. A study into the regularity of source code natural language artifacts reveals that natural language artifacts in source code have a much lower entropy, or unpredictability, when compared to more natural language [39]. All of these point towards the conclusion that there are significant differences between natural language and the artifacts found within source code.

Some research into tagging natural language artifacts in source code has been conducted [19, 29, 3]. This research has focused on developing taggers that can effectively parse and label natural language artifacts within source code, including

analysis of how words are used within specific software source code. However, this research has so far not conclusively demonstrated which taggers are most effective at recognizing which taggers are most accurate on source code artifacts. In one study, Sridhara et al. measured the effectiveness of existing natural language processing tools on source code artifacts. In this study, the focus was on semantic similarity, rather than on part-of-speech tagging, but it shows the inadequacy of many existing linguistic tools for source code language [38].

There are also quantifiable differences between source code parts of speech and the parts of speech that can be found in natural language writings, beyond simple differences in structure. AlSuhaibani demonstrated that there are differences between the parts of speech in source code, and that simple heuristics for identifying the parts of speech in source code (such as saying all method calls are verbs, and all fields are nouns), while not wholly inaccurate, ignores a large amount of information that can be derived from the source code [4]. This sort of information is the type that will be useful for generating summaries about lines of source code. AlSuhaubani creates a custom set of rules for parts of speech that is primarily reliant upon the function of a given word in a piece of code (e.g., an identifier is a verb if it makes some meaningful modification to some object or value within its scope). This difference does have an impact on the effectiveness of linguistic tools on source code artifacts [38, 22].

The usage of POS taggers and linguistic information in regards to source code is a well-established trend in various fields of software engineering. The usage of lingusitic information has been demonstrated to be useful for various purposes. For example, the usage of words has helped with source code searching [1, 15, 20, 36], program

comprehension [2, 6, 12], and error and bug reporting [40]. Despite this prevalence, to this point there has not been an exploratory study into how effective existing part-of-speech taggers are on source code. This is partially due to the limited number of taggers designed for labeling the part-of-speech of source code artifacts [19, 15].

2.3 Comment Generation

As was mentioned previously, comments in source code can be important to help a programmer to understand the function and implementation of a piece of code. However, documenting code with comments is a time-consuming process, and is often expensive, due to the required skill to understand the code to be able to document it, and since this documentation does not directly affect execution of the software, maintenance of software is often viewed as less important. However, ignoring the need for documentation can also lead to expense down the line, as maintenance of code that is poorly documented takes far longer, so in both cases expense is an issue [30]. For this reason, research has been performed exploring how to generate summaries and documentation for source code automatically. Some of this research, carried out by Ying and Robillard, has focused on the ways in which programmers most effectively summarize and explain code to an unfamiliar audience. The changes that were made included summarizing, changing formatting, and shortening code when possible, in order to make these changes as clear as possible [45]. This research was related to earlier work which tried to generate these sorts of explanatory statements automatically, based on both textual evidence and control flow of the program [16].

Others, including Rodeghero et al. have carried out eye tracking studies in order to determine what sections of code programmers look at when trying to determine what a section of code does. The research indicates that programmers primarily look at methods and field names, rather than control flow in order to understand what a piece of code does [33]. These results reinforce the ideas presented in Liblit's research, because the sections of code that an experienced programmer looks at are likely to be those with the most information, and will filter out the information that is less important [28].

Two particular research areas contribute extensively to what this thesis has explored. First, Sridhara, et al. conducted a study into how summaries can be generated from Java method signatures. Their solution combined the ideas of software analysis and natural language processing to generate these summaries [37]. Natural language processing, such as parsing method names for information, is closely tied to the part-of-speech tagging research conducted by others. However, in Sridhara's paper, different preexisting POS taggers were not used, and several assumptions were made about the most accurate taggers, which this thesis tests. Furthermore, this thesis differs from the work of Sridhara, et al. in that the previous work only performed summaries for method creation, while this work will summarize any line of code, and will do so for individual lines as opposed to generating summarizing comments for a whole method. Secondly, it has been shown that novice programers need different types of information in summaries than more advanced programmers, and therefore novice programmers will not benefit as much from the generation of comments that are intended for more experienced software developers [37]. This thesis will tailor our

summaries to specifically fit the needs of novice programmers, further distinguishing this thesis from the work that has come before.

2.4 Natural Language Artifacts

The second major contributor to this thesis is the work of Hill, et al. into natural language artifacts within software source code [20, 21]. Hill's work focuses on parsing and pulling information from source code based on the natural language artifacts found within it. This work focuses heavily on understanding the relationships between words in source code, allowing built-in search tools to return the most accurate results possible to the user. This is accomplished by treating words in relation to each other, not simply as a series of unrelated words. This work focused on searching, not on summaries of code. However, these papers did contribute to research on POS tagging, as they built upon the notion of the usefulness of natural language analysis in source code. Hill's work formed the foundation for the summary generation technique used by Sridhara et al. in summarizing source code [19]. Of particular note is Hill's creation of a "Software Word Usage Model" (SWUM), that allowed for the study and analysis of syntactic part-of-speech tagging of words in source code.

Summarization of source code has also been performed on Python, turning source code into "pseudocode", which is a form of abbreviated, simplified, and informal language that is designed to be easily understood. Similar to this thesis, this work summarizes code in a line-by-line manner, and generates statements that can be read and understood by a novice programmer. However, the pseudocode summary works

only on Python, while in this thesis, we have focused on Java. Additionally, the python pseudocode summarizer works primarily by parsing the abstract syntax tree (AST) of the compiled code, and relies less on parsing variable and method names to determine the functionality of the code [32, 13], whereas the research presented in this thesis relies on both.

Similarly, work has been done to automatically generate comments for source code that summarize and clarify the code. The lack of extensive comments in existing software repositories is a known issue, and as such work has been conducted into generating comments for source code. These efforts are intended to save time and money for those working on software projects, because adequate comments can help to clarify what source code does, but developing good documentation requires time of developers, so automation of the process could be a cost saving measure [37] . An example of work in comment generation is “Autocomment”, developed by Wong, Yang, and Tan. This software parses source code and uses its findings to generate comments and summarize the workings of source code, in much the same way that this thesis approaches the problem of summarizing source code [44]. The difference between Wong, Yang, and Tan’s work and this thesis is that their comments are mostly geared towards experienced programmers, and not novices, and therefore do less line-by-line summarization, and more big-picture summarization.

2.5 Program Tutors

In addition to programs which summarize and annotate source code in comments, some programs have been developed to teach an individual how to write code. These tools provide a useful service to students by providing guidance for what needs to be done to proceed with the creation of software. These tutors have been demonstrated to be effective in multiple ways, such as providing quick feedback, and improving students' confidence [14]. However, this thesis will make contributions to this idea in other ways. First, these tutors generally focus on the process of writing novel code, rather than learning to read existing code, which is another part of learning coding. Additionally, the software developed as part of this thesis will allow users to see a brief summary of existing code to gain better understanding of working with code.

Many have proposed developing tools which teach debugging, as this is a difficult skill to learn, especially in large classrooms where individualized attention might not be sufficient for students [8]. However, these tools, while effective for teaching how to write new code, are generally not used to teach students how to read code, particularly code written by others, which is an important part of learning to be an effective programer, due to the collaborative nature of most, if not all, major software projects.

Additionally, a variety of tools exist that have tried to improve debugging and approaches to software engineering. For example, Ko and Myers developed a tool which allowed programmers to ask questions about what output to expect from program execution, to allow for easier debugging. This leverages a variety of automated analysis, similar to the aim of this thesis [27]. This was a continuation of related work

which limited the scope to specific events and commands [31]. More recent research has been conducted comparing the AST of a student's program to that of an instructor's demonstration program to help create explanations of steps for the student to follow [47]. Furthermore, other research has been conducted on how to make compiler error notifications more easily comprehensible, which can be a barrier to novice programmers learning to identify errors in their own code [5]. This is related to this thesis in that both of these forms of research will aim to make the process of learning to read and write code easier for a novice programmer, though the methodologies are different.

In this thesis, we propose novel research that allows novices to work with production level code. This can teach best programming principles and practices from those with more experience. Additionally, these other program tutors focus generally on writing new code, as opposed to working with existing code, which is an additional task which can be difficult for a student to learn, and does not necessarily follow from the ability to write and think out novel code.

Chapter 3

Automatically Generating Explanations of Source Code for Novice Programmers

Processing source code to generate informative summaries requires several steps. The first of these is to find and summarize a variety of lines of Java source code. Then it is necessary to generate more generic summaries of lines based on the type of statement in the code. Then, once this has been accomplished, it is necessary to develop a way to parse the source code of a project in order to figure out the type of statement that was represented by each line of code. From there, using both the generic templates generated previously, and the part-of-speech tagger, explanations of lines of code can be made.

As part of this thesis, a tool was created called "Code Teacher", which is designed to provide information about what individual lines of code in a project do. Before

Code Teacher was implemented however, it was necessary to choose how to design it. A large variety of different tools and frameworks exist for software development and coding. Early in the process it was decided that Code Teacher would be designed to extend functionality of the Eclipse integrated development environment (IDE). There were several reasons for this choice. The major reason was that by leveraging existing functionality in the IDE, we could focus on the more novel ideas of the code. Additionally, the Eclipse IDE is widely used, making the evaluation of Code Teacher easier.

The final methodology for how to generate these types of summaries is as follows:

1. Collected six open source projects.
2. Randomly selected lines from source code.
3. Generated specific line summaries.
4. Generalized specific examples into templates.

3.1 Selection of Data for Analysis

To begin the process of creating summaries of source code, it was first necessary to find examples of different kinds of source code. This served two purposes. It first allowed us to conduct research into the more common types of statements that appear in production level source code. Additionally, it would allow us to examine source code to determine what kinds of metadata information in source code is helpful for the generation of summaries.

To begin with, we sought out new examples that had not been considered elsewhere in the study. To do this, we went on Github, and found several open source Java projects [26, 23, 24, 25]. The first of these projects, **Anthelion**, is a plugin for a piece of Apache software which parses HTML to crawl semantic annotations within the page [26]. The second project, **Slide**, is an open-source browser for the site reddit, built as an android app [23]. **Timber**, a third project, is an open source music/media player for android devices, built by integrating several existing tools [24]. **Plaid** was the final open source project that was used for this thesis. Plaid is a tool designed for development of user interfaces [25]. These tools were chosen at random to reflect a variety of different software tools, infrastructures, and design architectures, and therefore reflect a diversity of design tools.

These projects were selected by searching for open source repositories of code on Github. The selected repositories were listed as "trending", indicating that these were popular projects. These projects all employed production-level coding practices. After selecting several of these projects, individual files were selected at random. Then we selected random individual lines from each file, limiting the number of examples from each file to five, with three files from each project. Eventually, however, in order to reflect the variety of source code lines that appear in code, we searched for the types of code statements within other files so that we could find more complete examples. To protect the validity of these results, randomness was still utilized in the selection process.

3.2 Summary Overviews

Once these sample summaries were generated, it became necessary to generalize the summaries. Once this was done, it would be possible to program these general summaries into a summary generation methodology. Using the most accurate POS tagger, it would be possible to gather information from words in the code to supplement these summaries with additional information, and make them better matches for the specific instance of the code. For a list of these generic phrases, see 3.2.1.

The process for generating generic template summaries involved three steps. First, examples of each type of statement were collected. Then, several summaries were compared, and a more generic template was created. Finally, a sample template was made with variables that contained info that could be extracted from the original example of the code. Below is a list of all of the templates for types of source code statements. These statements assume that the part-of-speech information from the method names is unavailable.

- **Exception Handler:** This type of code statement determines how a program will handle actions which throw exceptions in the event of runtime errors. Therefore, it was most important to explain that this type of code will execute if an exception is thrown, and what type of exception will cause its execution. For this reason, the template which was created was "Executes code if an error (exception) of type [ExceptionType] occurs"

- **Generic Exception Handler:** For purposes of simplicity and clarity, this type of handler, where the exception is of type "exception" was separated from the

Example	Generic Summary from Example	template match	template summary
`} catch (final IllegalStateException ignored) {`	Executes code if an Illegal State Exception occurs	`} catch ([ExceptionType] [ExceptionName])`	Exectutes code if a(n) [ExceptionType] exception occurs
`throw new IllegalArgumentException(e);`	Raises exception due to an illegal arguement	`throw new [ExceptionType]([ExceptionName])`	Raises exception [ExceptionName] due to [ExceptionType]
`break;`	stops execution of a loop	`break;`	Stops execution of a loop
`} catch (Exception e) {`	Executes code if an exception of any kind occurs	`} catch (Exception [ExceptionName])`	Executes code if any exception occurs
`if (mService != null) {`	Indicates that the following code will execute if and only if mService is not null	`if (booleanCondition){`	The following code will execute only if [booleanCondition] is true.
`mService.setLockscreenAlbumArt(enabled)`	Sets the lockscreen album art of mService variable	`[object].[method]([Arguements])`	Executes [method] on the [object] variable
`public static boolean verifyPurchase(String base64PublicKey, String signedData, String signature) {`	Declares a method that will verify purchases	`[returnType] [methodName]([Arguments])`	Declares a method that will do [method Name] and return [returnType]
`return url;`	Returns the variable url to the calling method	`return [varName]`	Returns the variable [varName] to the calling method
`private int scrollingChildId = -1;`	declares an integer variable that describes the scrolling child id	`public/private [varType] [varName] = [value];`	Declares a variable of type [varType] that stores the value for [varName]
`mWrappedContext = context;`	Modifies the value of mWrappedContext to match the variable context	`[varName] = [varName2]`	Modifies the value of [varName] to be equal to [varName2]
`for (int i = 0; i < queue.size(); i++) {`	Executes the following code once for values of i between 0 and queue.size(), where i increases by 1 each execution	`for ([variable]=[number];[variable][comparator][value];[variable][valueChange]):`	Executes the following code once for values of [variable] between [number] and [value] where [variable] (increases/decreases) by [valueChange] each execution
`while (hasMore) {`	Executes the following code until hasMore is false	`while ([boolean]){`	Executes the following code until [boolean] is false
`do { .... } while(true);`	Execute the following code at least 1 time, and then if true is true, continue to execute until true is false	`do { .... } while [boolean]`	Execute code once, and then if [boolean] is true, continue to execute until [boolean] is false

FIGURE 3.2.1: Table With Examples of Generic Line Summaries

previous example. This summary template was "Executes code if an exception (error) of any kind occurs"

- **Exception Thrower:** Similarly to the exception handlers, the most relevant information for this type of code summary was the type and name of the exception being thrown, and an explanation of the fact that this code raised an error. The template for this statement was "Raises an exception (error) [ExceptionName] due to [ExceptionType]". The inclusion of the word error in the previous summaries was included to introduce the concept of what an exception is in clearer language to someone unfamiliar with the concept.

- **Break:** This type of statement was fairly unambiguous, and therefore relied very little on extra information from within the code. Because this type of statement is simple, a simple summary was needed. The template which we generated was "Stops execution of a loop".

- **If Statement:** This was a simple example of flow control within a program. The most important part for a summary of this line of code was getting the boolean condition for when a line of code will execute. The generic template for this kind of statement was "Executes the following code only if [booleanCondition] is true".

- **Method Call:** Without POS information, this type of statement is hard to summarize. In this case, all that can be done is to assume that the method name adequately describes what the method does in its default state. Without

POS information, the best summary we can provide is "Executes [method] (on [object])"

- **Method Declaration:** Similarly to a method call, a more successful implementation of such a summary would require POS information about the method being declared. Without this information, the summary of such a statement would be "Declares a method that will do [method Name] and return [returnType]"

- **Return Statement:** This type of statement determines what would be passed from a method to the method that called it. These statements determine how different methods "communicate" with each other. A good summary of this type of statement is "Returns the variable [varName] to the calling method"

- **Variable Declaration:** The summary for how a variable declaration would appear is one which could benefit from POS tagger information, to determine what a variable is used for. However, for this thesis, the emphasis was on method tagging, and not variable name tagging. An example of a summary without this information would be "Declares a variable of type [varType] that stores the value for [varName]"

- **Variable Value Modifier:** Similarly to the above example, POS information could hypothetically help with generating a more complete summary. However, without this information, a summary can still be made. Specifically, for this thesis, the generated summary was "Modifies the value of [varName] to be equal to [varName2]"

- **For Loop:** A for loop is generally used for iterating over a series of values, and doing something for each value. A summary of such a statement would be "Executes the following code once for values of [variable] between [number] and [value] where [variable] (increases/decreases) by [valueChange] each execution"

- **While Loop:** A while loop does something as many times as necessary until a specified condition is met. It may not execute if the condition is met before the first execution of the loop in the code. A summary of this type of line of code is "Executes the following code until [boolean] is false"

- **Do-While Loop:** A do-while loop is similar to a while loop, except that a do-while loop is guaranteed to run at least once, while a while loop may not execute, depending on initial conditions. The summary for this type of statement is "Execute code once, and then if [boolean] is true, continue to execute until [boolean] is false"

3.3 Programmatically Summarizing Source Code

After the common types of statements that appear in source code were summarized, it then became necessary to actually generate summaries of arbitrary lines of code. In order to do this, we leveraged parts of the Eclipse development environment, and extended these as necessary.

The method of implementing a source code summarizer first required becoming acquainted with Eclipse's infrastructure, to see how it was possible to extend and

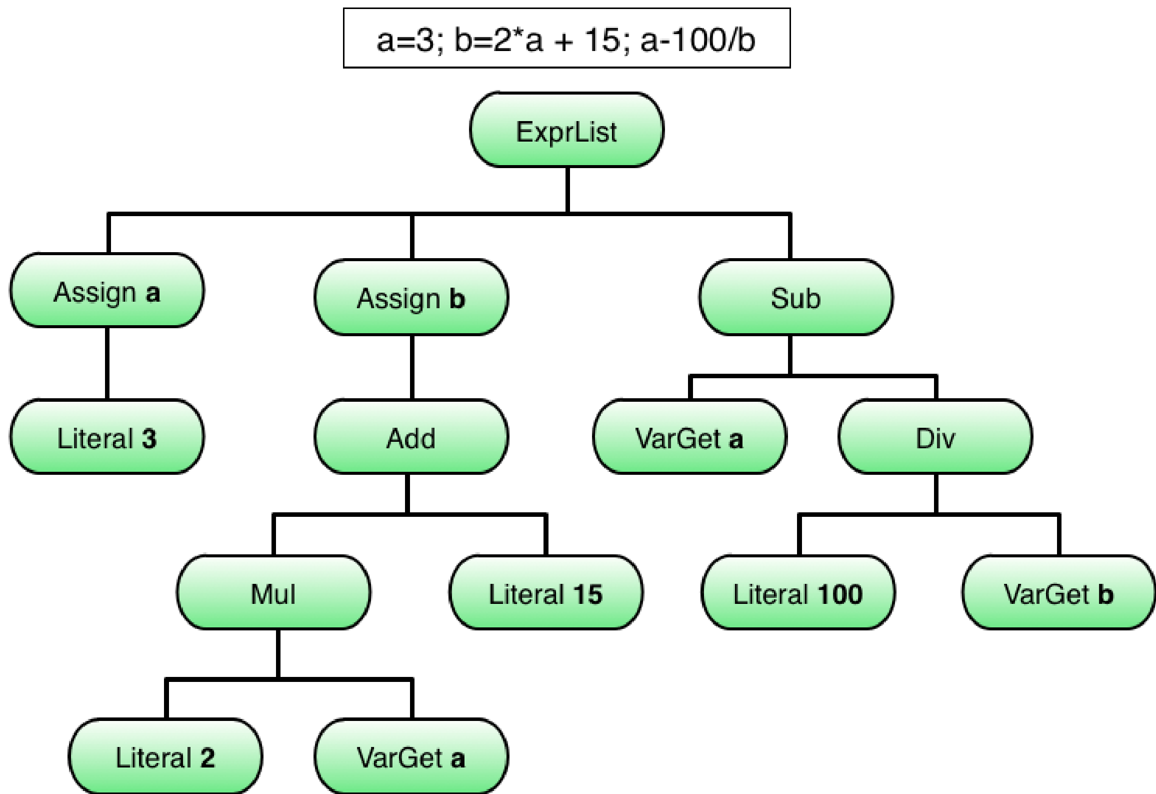

FIGURE 3.3.1: AST example

modify its default functionality. Ultimately, the decision was made to make the software work as a custom editor that would execute independently of the default editor environment. One challenge in this decision was attempting to learn how to integrate the behavior and interface of the default eclipse editor to make the interface as intuitive as possible for a new user, as well as for someone more familiar with the default eclipse editor's functionality.

The next challenge was determining what type of statement a given line of code was. For example, a line of code may be an "if" statement, or a variable declaration, or a method call. All of these different types of code would need to be summarized

differently from each other. In order to determine what a given section of code was, we extended the default editor's tools to determine a similar issue. Most of our changes were made by extending the default ASTVisitor class, a class which has methods for parsing and traversing the AST of arbitrary java source code files. For an example of an AST, see figure 3.3.1. This figure was found at http://leanovate.github.io/bedcon/talk/abstract_syntax_tree.html. At the top of this figure, there can be seen three java-type statements. On the far left, two nodes can be seen which are a representation of the first statement, an assignment of a literal value “3” to a variable “a”. This is the type of information which is parsed by the software to collect information about the execution of code.

3.4 Tool Design and Requirements

Code Teacher is developed as a plugin for the Eclipse IDE, designed as a custom java editor. This means the only requirements for running this software are that the user have a working version of eclipse, and that, in Code Teacher's current state, the files which are to be worked with are Java files with a .jav extension, as opposed to the more common .java extension. The editor works by replacing the default pop-up summaries of the normal java editor with ones that are designed to be easily comprehensible and useful to novice programmers.

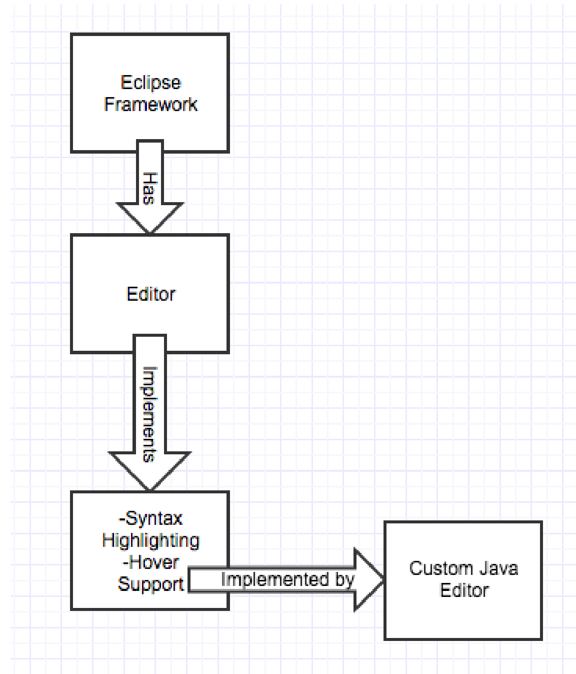

FIGURE 3.4.1: Simple overview of Eclipse's architecture

3.4.1 Extension Points Within Eclipse

Code Teacher was primarily implemented as a new class, called SummaryVisitor, which provides the functionality to generate the summaries of source code by visiting nodes of the AST of the source code. This is an extension of the built-in ASTVisitor class, and overwrites many of the built-in methods of that class.

The ASTVisitor class was our primary hook into the default environment of Eclipse. By creating a new instance of the SummaryVisitor class that extended the ASTVisitor class, and by creating a new instance of this class within the JavaTextHover class of the custom editor. Specifically, the instance was created and returns a string to the getHoverInfo2 method, which then returns the string to the method responsible for

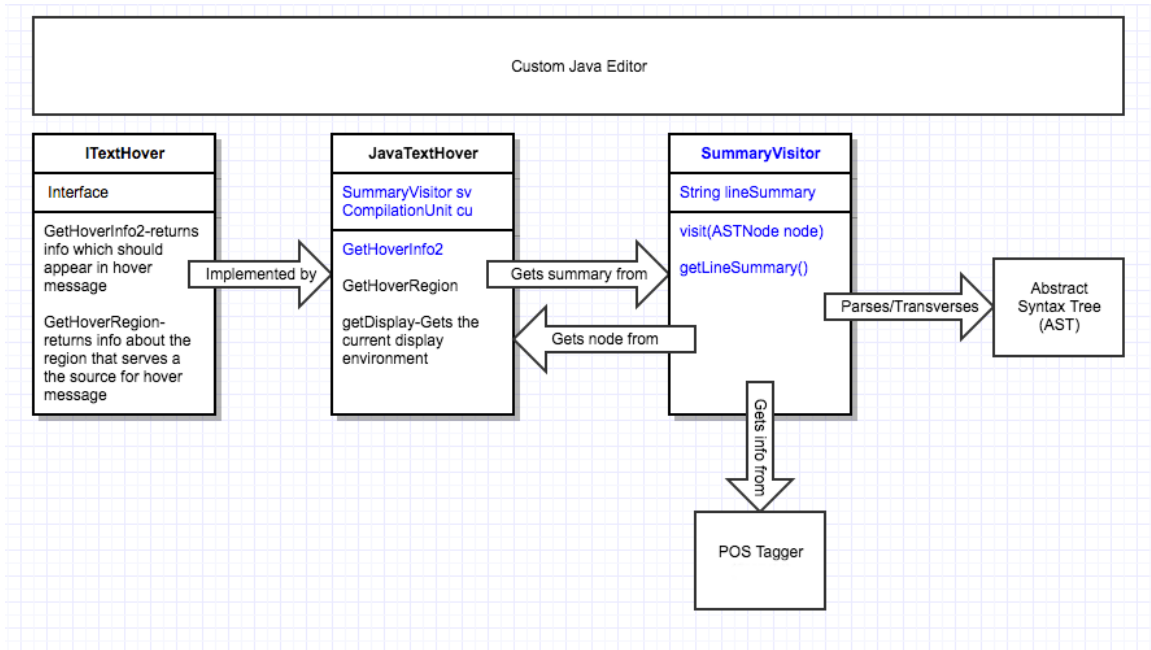

FIGURE 3.4.2: In depth summary of the hover support methodology

rendering the string to the screen. The SummaryVisitor class extends ASTVisitor, with specialized visit methods for many different types of AST Nodes. These methods return true if the traversal of the tree should continue downward, and false if the traversal should stop at the given type of node.

3.4.2 Overview of Eclipse Architecture

For a high-level overview of how this tool was developed, see figure 3.4.1. As is shown in this image, at the foundation of Code Teacher is the existing Eclipse architecture, which provides a variety of functionality, such as hierarchical organization, and rendering. However, for this tool, much of this is treated as a “black box”, where the

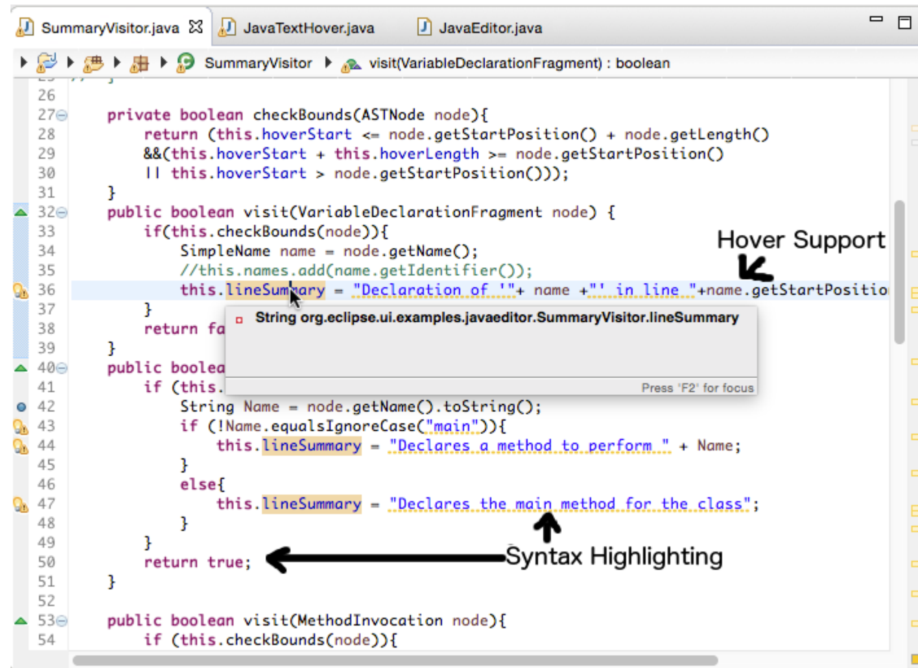

FIGURE 3.4.3: Illustration of functionality provided by the default eclipse editor

implementation is unimportant. The only relevant pieces of the code are the ones which are responsible for implementing the editors. Editors, in turn, implement a variety of functionality for the eclipse application, such as syntax highlighting, and hover support. Figure 3.4.3 illustrates the types of functionality implemented by the default eclipse editor.

At the end of the diagram in figure 3.4.1, the eclipse editor is implemented by a custom editor. The infrastructure of this custom editor, and how a hover summary is generated is shown in figure 3.4.2. The ITextHover interface is at the base of this functionality. The implementation of this interface is what allows for a hover annotation to be rendered to the page. However, what is displayed on the screen is itself implemented by a series of calls to objects and instances of other classes, most

```
MusicPlayer.jav
        try {
            if (mService != null) {
                return mService.removeTrack(id);
            }
        } catch (final RemoteException ingored) {
        }
        return 0;
    }

    public static final boolean removeTrackAtPosition(final long id, final int position) {
        try {
            if (mService != null) {
                return mService.removeTrackAtPosition(id, position);
            }
        } catch (final RemoteException ingored) {
        }
        return false;
    }

    public static void moveQueueItem(final int from, final int to) {
        try {
          Execute the following code, with acknowledgement that there may be an error
                mService.moveQueueItem(from, to);
            } else {
            }
        } catch (final RemoteException ignored) {
        }
    }

    public static void playArtist(final Context context, final long artistId, int position,
        final long[] artistList = getSongListForArtist(context, artistId);
```

FIGURE 3.4.4: Demonstration of "try" summarization

notably the custom summary visitor class. In this figure, all custom implemented or modified methods and fields are highlighted in blue text. The elements in black are either extended from default Eclipse implementation, such as with ASTs, or integrated from other programs, as is the case with POS tagging.

3.4.3 Using Code Teacher

The source code for Code Teacher is available in a public git repository, found at `https://bitbucket.org/wyatt-olney/codeteacher` (see Appendix A for further information about where the modified code can be found). Additionally, in figures 3.4.4 3.4.5, 3.4.6 and 3.4.7, runtime execution screenshots of Code Teacher can be seen. The code used in these demonstrations all come from **timber** [24].

In figures 3.4.4, 3.4.5 and 3.4.6, Code Teacher is working as intended without the

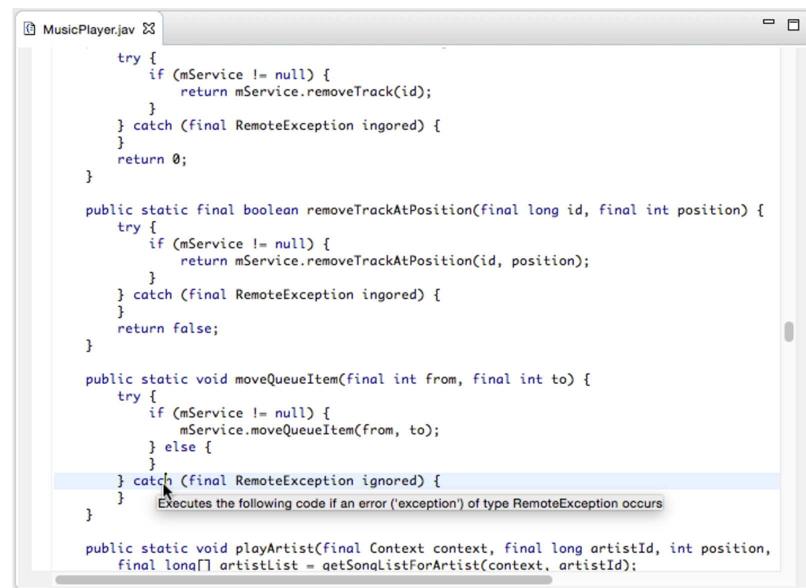

Figure 3.4.5: Demonstration of “catch” summarization

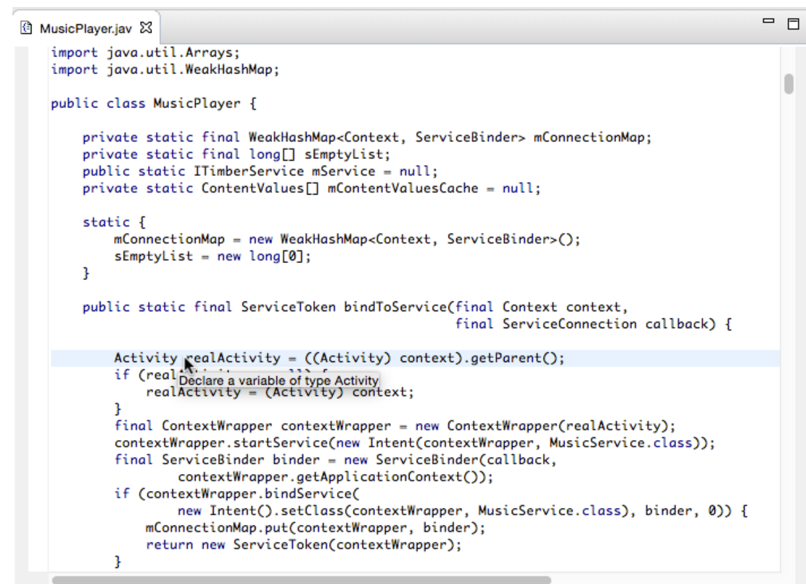

Figure 3.4.6: Demonstration of variable declarations

FIGURE 3.4.7: Demonstration of method summarization

need for POS information. However, the inclusion of such information could improve the summaries that are generated by Code Teacher. In figure 3.4.7, Code Teacher attempts to summarize a method call, which is possible, but often uninformative and sub-optimal, without part-of-speech information.

Code Teacher provides functionality and instruction that could prove helpful to a novice programmer. For example, a novice may not know the purpose of a "try/catch" statement. Summaries such as the ones presented in figures 3.4.4 and 3.4.5 may provide a partial, brief explanation of what a given line of code does. Additionally, especially in an object oriented language such as Java, variable declarations may be confusing for novices. Distinguishing between the variable name, and the class name may be a point of confusion for novices. The summary presented in figure 3.4.6 can help to clarify some of this confusion, by explaining which part of the line is the class.

3.5 Inclusion of POS information

Currently there is a limitation to “Code Teacher” in that it currently does not support integration of POS info. While summaries of lines of source code exist within the current framework, these are not the optimal summaries, as they consist only of simple templates, without integration of POS info.

The best way to illustrate the usefulness of POS tagging in generating summaries for lines of source code would be to illustrate how this type of information could benefit a summary of source code. For an example of how this information could be used, consider the method isPlaybackServiceEnabled, from **timber** [24]. Without POS information, the summary for such a method declaration would be “Declares a method that will perform isPlaybackServiceConnected and return a boolean”. This is both grammatically awkward, and not particularly informative, nor is it useful for determining what the method actually does.

However, with POS information being included, a more useful summary can be generated. Specifically, the summary utilizing POS information on this line of code would be “Declares a method that checks if the playback service is connected, and returns a boolean”. In addition to being more colloquial, this also is a more informative summary of the method declaration presented here.

Chapter 4

POS Tagging Identifiers in Source Code

As has been discussed previously, while a variety of different part-of-speech taggers exist and are used frequently, there has not been a comprehensive study of the accuracy of these taggers on the unique corpus of source code identifiers. For this thesis, the accuracy of taggers on this corpus is essential to creating the most accurate summarizations of lines of source code. For this reason, this thesis presents novel research that explores the effectiveness of existing part-of-speech taggers on how they work on source code identifiers, specifically method names.

The study presented here has several parts, explained in greater detail below. The steps involved in designing and completing the experiment are:

1. Selection of the most relevant POS taggers.
2. Creation and manual labelling of a "gold set", or set of source code identi-

fiers which provide us with a consistent standard for comparison which we can compare all POS taggers against.

3. Implementation of each individual POS tagger, and execution on the gold set.

4. Evaluating the accuracy of each POS tagger, including unification of different tagsets.

4.1 Selection of Taggers

Due to the large number of different taggers, it became clear early on that it would be necessary to evaluate only the most relevant and common part-of-speech taggers for our experiment. The selection process took into consideration several different factors, including reported accuracy of the tagger (usually on the Wall Street Journal Corpus, a common natural language corpus used for evaluating taggers and other linguistic research), the frequency with which the tagger is compared to by other taggers, and the corpus which the tagger was trained on and what the tagger was designed to do.

The methodology that we used to select the most relevant taggers was based on several factors. First among these was apparent relevancy within the research community. This was usually based on which taggers were frequently examined or compared to. For this thesis, it was sufficient for our purposes to conclude that if a particular tagger was compared against frequently, it was one for which a general consensus existed within the research community as to its effectiveness on traditional

natural language sources. This thesis would test this effectiveness on source code artifacts.

A secondary criterion that was used to determine if a tagger was going to be worth testing was the corpus for which it was designed. As has been mentioned before, most taggers which we looked at were trained and developed for a natural language corpus. Most frequently, the corpus which taggers used for evaluation was the Wall Street Journal corpus. These taggers were not ruled out of our study, and many of these were used. However, other corpuses and training sets were used for other taggers. Two taggers in particular were developed for source code POS tagging specifically, so those were used [19, 29]. Additionally, taggers which were developed to work on corpuses that had frequent abbreviations and other similarities to source code were used [11]. However, other taggers were excluded, because their intended corpus was too dissimilar from that of source code to be worth investigating [43].

In the following list, an explanation of why each tagger was selected to be evaluated is presented, along with some basic information about each tagger's reported accuracy and other pertinent information, such as the approach used in tagging, where applicable.

- **Stanford Tagger:** This tagger was selected for evaluation because it was the tagger that was most commonly compared to by other work in the field of natural language processing. On the corpuses that it was tested on, the Stanford Tagger had a reported accuracy of 97% [41], which is high compared to many other taggers. It has also been applied to field names and found to have 88% accuracy [6] and 90% accuracy on bug reports [40]

- **GATE's Twitter Tagger:** The Twitter Tagger was designed to work with sparse and noisy data [11]. While this tagger was designed to parse and label twitter messages, the nature of that when compared to source code had some similarities (e.g. frequent abbreviations, loose grammatical rules). The tagger has a reported accuracy of approximately 91%.

- **RASP:** This tagger had one of the lower reported accuracies among taggers that we had tested, with a reported accuracy of 76.3% [7]. However, it was a tagger that was still fairly often compared against, and has been used in prior applications of POS tagging to source code [46].

- **Apache OpenNLP:** OpenNLP is an open source POS tagger that uses a maximum entropy approach to tag the words.

- **SpaCy:** SpaCy was written in Python, unlike many other taggers, and is competitive with other taggers in terms of accuracy. Its reported accuracy was evaluated to be 91.8% on a natural language corpus [9]. SpaCy was also a more recently developed tagger.

- **University of Pennsylvania's LTAG-Spinal Tagger:** The LTAG-Spinal tagger was reported to have an accuracy of 97.33% on the Penn Treebank corpus [34]. Additionally, this tagger uses a unique algorithm for developing its part-of-speech tags.

- **POSSE:** This was one of the two part-of-speech taggers that was developed to label and parse the parts of speech of source code artifacts, and draws informa-

tion from the source code to help its labeling [29]. On the corpus that it was originally tested upon, there was an effective accuracy of 192 out of 196 phrases parsed correctly (98%).

- **SWUM:** This part-of-speech tagger was designed to be effective at part-of-speech tagging on source code language artifacts [19]. It was designed by Hill, and uses a variety of contextual information from source code to inform its decision making process for POS tagging. It later became the basis for the work of Malik in developing the POSSE tagger [29].

- **Stanford, I-Prepended:** In addition to running each tagger, a secondary experiment was run in order to see if slight, systematic alteration of each tagger would yield better results. Specifically, for one tagger, Stanford, a secondary trial was run, where the word "I" was prepended before each identifier's name. This was done to test to see if mimicking normal english syntax would improve the accuracy of the taggers. This approach has also been used to parse source code identifiers [36].

4.2 Experimental Gold Set

After selecting taggers, the next step was to evaluate the effectiveness of all of the taggers. However, before we could do so, we needed to create a "gold set", or a collection of examples from the corpus for which the tagger is to be evaluated. For the purpose of this thesis, the gold set is a collection of source code identifiers. A

gold set is required to be tagged manually by a human, and these manual tags are assumed to be correct, so that any other taggers can be compared to this set. This creates a point of comparison for all of the existing taggers.

For this thesis, the gold set consists of 195 identifiers from source code, all function names taken from various open-source, publicly available Java projects. This gold set was designed and explained in greater depth by Sana Malik [29]. This was considered to be a random, representative sample collected from 22 different open source software projects.

Additionally, after an initial study of the accuracy of taggers, a secondary testing set was created for the purpose of evaluating the most effective taggers. The reason for this was that the original gold set, which was used for our initial evaluation, was used by Malik to train one of the taggers that was tested in this experiment. Therefore, a secondary testing set was needed, in order to get a fair evaluation of the taggers. For this, we selected an additional 41 method names from the source code of four open-source projects. These were then parsed and annotated manually, and then were used as input for the taggers that had been evaluated to be the most effective in our initial study of method identifiers.

This gold set's design is important for the purpose of this study because it helps to determine what kind of data will be produced. The choice to focus on method names does limit the scope of identifiers that were evaluated, because artifacts such as variable names are not taken into consideration by this gold set. However, as was discussed previously, research by Liblit, Begel and Sweetser indicated that more visible code is more likely to be informatively named than less visible elements of code

such as variable names [28]. This implies that local variables are less likely to have names which could be parsed for additional information, or return meaningful results. Additionally, according to AlSuhaibani, a simple heuristic rule such as labeling all variables as nouns may be reasonable [4].

4.3 Experiment Design

Broadly speaking, the central measure of how effective a given tagger was in labeling parts of speech was how closely the output of a tagger matched the annotation of the same phrase by human evaluators. The first task for this evaluation was unifying different tagsets. This is necessary because different taggers use different tagsets, which vary in robustness and the number of possible tags. However, the robustness of the tagset used by a tagger should not be a factor in how accurate a tagger is evaluated to be, so for this reason it becomes necessary to create a relation between the different tagsets, so that the tags correctly correspond to each other. For this thesis, the tagsets that had to be considered were the Penn Treebank tagset, the CLAWS7 tagset, and the tagset that was proposed by Hill for her “SWUM” tagger [19]. A table with a summary of the unification of the different tagsets is presented in figure 4.3.1. Additionally, figures 4.3.1, 4.3.2, 4.3.3 and B.0.4 show more in-depth analysis and comparisons for tagset unification.

Once the experimental gold set was created, and the tagsets were unified, we ran the taggers on the corpus and compare the outputs to the original gold set. A good tagger will have a high frequency of matching tags compared to the gold set. There

Explanation	CLAWS7 Tag	CLAWS-SWUM Map	Gold Tag	GOLD-SWUM Map	Penn Tag	Penn-SWUM Map
possessive pronoun, pre-nominal (e.g. my, your, our)	APPGE	N	**Pronoun (PR)**	N	PRP$	PR
article (e.g. the, no)	AT	DT	**Determiner (DT)**	DT	DT	DT
singular article (e.g. a, an, every)	AT1	DT	**Determiner (DT)**	DT	DT	DT
before-clause marker (e.g. in order (that),in order (to))	BCL	DT	**Determiner (DT)**	DT	DT	DT
coordinating conjunction (e.g. and, or)	CC	CJ	**Conjunction (CJ)**	CJ	CC	CJ
adversative coordinating conjunction (but)	CCB	CJ	**Conjunction (CJ)**	CJ	CC	CJ
subordinating conjunction (e.g. if, because, unless, so, for)	CS	CJ	**Conjunction (CJ)**	CJ	CC	CJ
as (as conjunction)	CSA	CJ	**Conjunction (CJ)**	CJ	CC	CJ
than (as conjunction)	CSN	CJ	**Conjunction (CJ)**	CJ	CC	CJ
that (as conjunction)	CST	CJ	**Conjunction (CJ)**	CJ	CC	CJ
whether (as conjunction)	CSW	CJ	**Conjunction (CJ)**	CJ	CC	CJ
after-determiner or post-determiner capable of pronominal function (e.g. such, former, same)	DA	DT	**Determiner (DT)**	DT	DT	DT
singular after-determiner (e.g. little, much)	DA1	DT	**Determiner (DT)**	DT	DT	DT
plural after-determiner (e.g. few, several, many)	DA2	DT	**Determiner (DT)**	DT	DT	DT
comparative after-determiner (e.g. more, less, fewer)	DAR	DT	**Determiner (DT)**	DT	DT	DT
superlative after-determiner (e.g. most, least, fewest)	DAT	DT	**Determiner (DT)**	DT	DT	DT
before determiner or pre-determiner capable of pronominal function (all, half)	DB	DT	**Determiner (DT)**	DT	PDT	DT
plural before-determiner (both)	DB2	DT	**Determiner (DT)**	DT	PDT	DT
determiner (capable of pronominal function) (e.g any, some)	DD	DT	**Determiner (DT)**	DT	DT	DT
singular determiner (e.g. this, that, another)	DD1	DT	**Determiner (DT)**	DT	DT	DT
plural determiner (these,those)	DD2	DT	**Determiner (DT)**	DT	DT	DT
wh-determiner (which, what)	DDQ	DT	**Determiner (DT)**	DT	DT	DT
wh-determiner, genitive (whose)	DDQGE	DT	**Determiner (DT)**	DT	DT	DT
wh-ever determiner, (whichever, whatever)	DDQV	DT	**Determiner (DT)**	DT	DT	DT
existential there	EX	DT	**Determiner (DT)**	DT	EX	UN
formula	FO	UN	**Unknown**	UN		
unclassified word	FU	UN	**Unknown**	UN		
foreign word	FW	UN	**Unknown**	UN	FW	UN
for (as preposition)	IF	P	**Preposition (P)**	P	IN	P
general preposition	II	P	**Preposition (P)**	P	IN	P
of (as preposition)	IO	P	**Preposition (P)**	P	IN	P
with, without (as prepositions)	IW	P	**Preposition (P)**	P	IN	P
general adjective	JJ	NM	**ADJ**	NM	JJ	NM
general comparative adjective (e.g. older, better, stronger)	JJR	NM	**ADJ**	NM	JJR	NM
general superlative adjective (e.g. oldest, best, strongest)	JJT	NM	**ADJ**	NM	JJS	NM
catenative adjective (able in be able to, willing in be willing to)	JK	NM	**ADJ**	NM	JJ	NM
cardinal number,neutral for number (two, three..)	MC	D	**Digit (#)**	D	CD	D
singular cardinal number (one)	MC1	D	**Digit (#)**	D	CD	D
plural cardinal number (e.g. sixes, sevens)	MC2	D	**Digit (#)**	D	CD	D
genitive cardinal number, neutral for number (two's, 100's)	MCGE	D	**Digit (#)**	D	CD	D
hyphenated number (40-50, 1770-1827)	MCMC	D	**Digit (#)**	D	CD	D
ordinal number (e.g. first, second, next, last)	MD	D	**Digit (#)**	D	CD	D
fraction,neutral for number (e.g. quarters, two-thirds)	MF	D	**Digit (#)**	D	CD	D
singular noun of direction (e.g. north, southeast)	ND1	N	**Noun (N)**	N	NN	N
common noun, neutral for number (e.g. sheep, cod, headquarters)	NN	N	**Noun (N)**	N	NN	N
singular common noun (e.g. book, girl)	NN1	N	**Noun (N)**	N	NN	N
plural common noun (e.g. books, girls)	NN2	NP	**Plural Noun (NP)**	NP	NNS	**NP**
following noun of title (e.g. M.A.)	NNA	N	**Noun (N)**	N	NN	N
preceding noun of title (e.g. Mr., Prof.)	NNB	N	**Noun (N)**	N	NN	N
singular locative noun (e.g. Island, Street)	NNL1	N	**Noun (N)**	N	NN	N
plural locative noun (e.g. Islands, Streets)	NNL2	NP	**Plural Noun (NP)**	NP	NNS	**NP**
numeral noun, neutral for number (e.g. dozen, hundred)	NNO	N	**Noun (N)**	N	NN	N
numeral noun, plural (e.g. hundreds, thousands)	NNO2	**NP**	**Plural Noun (NP)**	NP	NNS	**NP**
temporal noun, singular (e.g. day, week, year)	NNT1	N	**Noun (N)**	N	NN	N
temporal noun, plural (e.g. days, weeks, years)	NNT2	**NP**	**Plural Noun (NP)**	NP	NNS	**NP**
unit of measurement, neutral for number (e.g. in, cc)	NNU	N	**Noun (N)**	N	NN	N
proper noun, neutral for number (e.g. IBM, Andes)	NP	N	**Noun (N)**	N	NN	N
singular proper noun (e.g. London, Jane, Frederick)	NP1	N	**Noun (N)**	N	NN	N
plural proper noun (e.g. Browns, Reagans, Koreas)	NP2	NP	**Plural Noun (NP)**	NP	NNS	**NP**
singular weekday noun (e.g. Sunday)	NPD1	N	**Noun (N)**	N	NN	N
plural weekday noun (e.g. Sundays)	NPD2	**NP**	**Plural Noun (NP)**	NP	NNS	**NP**
singular month noun (e.g. October)	NPM1	N	**Noun (N)**	N	NN	N
plural month noun (e.g. Octobers)	NPM2	NP	**Plural Noun (NP)**	NP	NNS	**NP**
locative adverb (e.g. alongside, forward)	RL	VM	**Adverb (ADV)**	VM	RB	VM
general adverb	RR	VM	**Adverb (ADV)**	VM	RB	VM
be, base form (finite i.e. imperative, subjunctive)	VB0	V	**Verb (V)**	V	VB	V
do, base form (finite)	VD0	V	**Verb (V)**	V	VB	V
have, base form (finite)	VH0	V	**Verb (V)**	V	VB	V
had (past tense)	VHD	PP	**Past tense verb (VP)**	PP	VBD	PP
having	VHG	V3PS	**Adjectival third person verb (AV3)**	V3PS		
had (past participle)	VHN	PP	**Past tense verb (VP)**	PP	VBD	PP
modal auxiliary (can, will, would, etc.)	VM	V	**Verb (V)**	V	MD	VI -> V
base form of lexical verb (e.g. give, work)	VV0	V	**Verb (V)**	V	VB	V
past tense of lexical verb (e.g. gave, worked)	VVD	PP	**Past tense verb (VP)**	PP	VBD	PP
-ing participle of lexical verb (e.g. giving, working)	VVG	V3PS	**Adjectival third person verb (AV3)**	V3PS	VBG	V3PS
past participle of lexical verb (e.g. given, worked)	VVN	PP	**Past tense verb (VP)**	PP	VBN	PP
-s form of lexical verb (e.g. gives, works)	VVZ	V	**Verb (V)**	V	VBD	PP
not, n't	XX	UN	**Abreviation**	UN		
singular letter of the alphabet (e.g. A,b)	ZZ1	UN	**Abreviation**	UN		

FIGURE 4.3.1: Unified Tagset

Table 4.3.1: CLAWS7 Tagset: Nouns

Tag	Definition	Gold Set Equivalent
MC	cardinal number (two, three...)	D
MC1	singular cardinal number (one)	D
MC2	plural cardinal number (sixes, sevens)	D
MCGE	genitive cardinal number (two's, 100's)	D
MCMC	hyphenated number (40-50, 1770-1827)	D
MD	ordinal number (first, second, next, last...)	D
MF	fraction (quarters, two-thirds)	D
AT	article (the, no)	DT
AT1	singular article (a, an, every)	DT
BCL	before-clause marker (in order to)	DT
DA	after-determiner (such, former, same)	DT
DA1	singular after-determiner (little, much)	DT
DA2	plural after-determiner (few, several, many)	DT
DAR	comparative after-determiner (more, fewer)	DT
DAT	superlative after-determiner (most, fewest)	DT
DB	before determiner or pre-determiner (all, half)	DT
DB2	plural before-determiner (both)	DT
DD	determiner (any, some)	DT
DD1	singular determiner (this, that, another)	DT
DD2	plural determiner (these,those)	DT
DDQ	wh-determiner (which, what)	DT
DDQGE	wh-determiner, genitive (whose)	DT
DDQV	wh-ever determiner (whichever, whatever)	DT
EX	existential there	DT
APPGE	possessive pronoun (my, your, our)	N
ND1	singular noun of direction (north, southeast)	N
NN	common noun (sheep, cod, headquarters)	N
NN1	singular common noun (book, girl)	N
NNA	following noun of title (M.A.)	N
NNB	preceding noun of title (Mr., Prof.)	N
NNL1	singular locative noun (Island, Street)	N
NNO	numeral noun (dozen, hundred)	N
NNT1	temporal noun, singular (day, week, year)	N
NNU	unit of measurement (in, ml)	N
NP	proper noun (IBM, Andes)	N
NP1	singular proper noun (London, Jane, Frederick)	N
NPD1	singular weekday noun (Sunday)	N
NPM1	singular month noun (October)	N
JB	attributive adjective (main, chief)	NM
JJ	general adjective	NM
JJR	comparative adjective (older, better, stronger)	NM
JJT	superlative adjective (oldest, best, strongest)	NM
JK	catenative adjective (able to, willing to)	NM
NN2	plural common noun (books, girls)	NP
NNL2	plural locative noun (Islands, Streets)	NP
NNO2	numeral noun, plural (hundreds, thousands)	NP
NNT2	temporal noun, plural (days, weeks, years)	NP
NP2	plural proper noun (Browns, Reagans, Koreas)	NP
NPD2	plural weekday noun (Sundays)	NP
NPM2	plural month noun (Octobers)	NP

Table 4.3.2: CLAWS7 Tagset: Verbs & Prepositions

Tag	**Definition**	**Gold Set Equivalent**
IF	for (as preposition)	P
II	general preposition	P
IO	of (as preposition)	P
IW	with, without (as prepositions)	P
VHD	had (past tense)	PP
VHN	had (past participle)	PP
VVD	past tense of lexical verb (gave, worked)	PP
VVN	past participle of lexical verb (given, worked)	PP
FO	formula	UN
FU	unclassified word	UN
FW	foreign word	UN
XX	not, n't	UN
ZZ1	singular letter of the alphabet (A, b)	UN
VB0	be, base form (finite imperative, subjunctive)	V
VD0	do, base form (finite)	V
VH0	have, base form (finite)	V
VM	modal auxiliary (can, will, would, etc.)	V
VV0	base form of lexical verb (give, work)	V
VVZ	-s form of lexical verb (gives, works)	V3
VHG	having	V3
VVG	-ing participle of lexical verb (giving, working)	V3
REX	appositional adverb (namely, e.g.)	VM
RL	locative adverb (alongside, forward)	VM
RR	general adverb	VM
CC	coordinating conjunction (and, or)	CJ
CCB	adversative coordinating conjunction (but)	CJ
CS	subordinating conjunction (if, because, so, for)	CJ
CSA	as (as conjunction)	CJ
CSN	than (as conjunction)	CJ
CST	that (as conjunction)	CJ
CSW	whether (as conjunction)	CJ

Table 4.3.3: Penn Treebank Tagset

Tag	**Definition**	**Gold Set Equivalent**
DT	Determiner	DT
PDT	Predeterminer	DT
WDT	Wh-determiner	DT
NN	Noun, singular or mass	N
NNP	Proper noun, singular	N
JJ	Adjective	NM
JJR	Adjective, comparative	NM
JJS	Adjective, superlative	NM
NNS	Noun, plural	NP
NNPS	Proper noun, plural	NP
PRP	Personal pronoun	PR
PRP$	Possessive pronoun	PR
WP$	Possessive wh-pronoun	PR
IN	Preposition or subordinating conjunction	P
TO	to	P
WP	Wh-pronoun	PN
VBD	Verb, past tense	PP
VBN	Verb, past participle	PP
MD	Modal	V
VB	Verb, base form	V
VBP	Verb, non-3rd person singular present	V
VBZ	Verb, 3rd person singular present	V3
VBG	Verb, gerund or present participle	V3
RB	Adverb	VM
RBR	Adverb, comparative	VM
RBS	Adverb, superlative	VM
WRB	Wh-adverb	VM
RP	Particle	VPR
CC	Coordinating conjunction	CJ
CD	Cardinal number	D
EX	Existential there	UN
FW	Foreign word	UN
LS	List item marker	UN
POS	Possessive ending	UN
SYM	Symbol	UN
UH	Interjection	UN

Table 4.3.4: Mapping part-of-speech tags across tagsets

Description	**Gold set tag**	**Penn Treebank**	**CLAWS7**	**SWUM/POSSE**
Noun	N	NN, NNP	NN, NN, NNA, NNB, NNL1, NPM1, ND1, APPGE	N, NI, NM
Plural Noun	NP	NNS, NNPS	NN2, NNL2, NNO2, NNT2, NP2, NPD2, NPM2	NP, NPL
Pronoun	PN	PRP, PRP$, WP$	PN, PN1, PNQO, PNQS, PNQV, PNX1, PPGE, PPH1, PPHO1, PPHO2, PPHS1, PPHS2, PPIO1, PPIO2, PPIS1, PPIS2, PPX1, PPX2, PPY	PN
Determiner	DT	DT, PDT, WDT	AT, AT1, BCL, DA, DA1, DA2, DAR, DAT, DB, DB2, DD, DD1, DD2, DDQ, DDQGE, DDQV, EX	DT
Adjective	ADJ	JJ, JJR, JJS	JB, JJ, JJR, JJT, JK	ADJ
Preposition	P	IN, TO	IF, II, IO, IW	P
Past Tense Verb	PP	VBD, VBN	VHD, VHN, VVD, VVN	PP
Verb	V	MD, VB, VBP	VB0, VD0, VH0, VM, VV0	V, VING, VB, VI
3rd Person Present Verb	V3	VBG, VBZ	VHG, VVG, VVZ	V3
Adverb	VM	RB, RBR, RBS, WRB	REX, RL, RR	VM
Verb Particle	VPR	RP	RP	VPR
Conjunction	CJ	CC	CC, CCB, CS, CSA, CSN, CST, CSW	CJ
Digit	D	CD	MC, MC1, MC2, MCGE, MCMC, MD, MF	D
Unknown	UN	EX, FW, LS, POS, SYM, UH	FO, FU, FW, XX, ZZ1	UN, ABV

are a few different aspects to consider when evaluating these taggers, mostly per-word accuracy and per-identifier accuracy. Per-word accuracy is how accurate a tagger is on any given word. Per-identifier accuracy is based on how often a tagger will accurately label a whole method name's part-of-speech tags. These two measures are related, but slightly different, so it is important to measure both factors to evaluate a tagger's accuracy.

4.4 Results & Analysis

After generating the two gold sets, running all of the selected taggers on both of them, then comparing outputs to the gold set, we analyzed the results of the taggers. For the purposes of comparing each tagger, we converted each tag, including the gold set tags, to their SWUM tagset equivalency, and then compared those equivalent tags. Here, we present box plots of the accuracy of each tagger. The mean for each of the taggers is denoted by the red plus, while the interquartile range is represented by the top and bottoms of the rectangles, with the median represented by the thicker, bold line. The taggers are listed in order of increasing mean.

Figures 4.4.1 and 4.4.2 show the per-word accuracy in the original gold set, and the second gold set respectively. These graphs indicate the accuracy of each word taken independent of what phrase they are in. The most accurate taggers for this section were POSSE for the POSSE gold set, while on the other gold set, the I-prepended Stanford tagger achieved the highest accuracy, despite the fact that it was the 4th most accurate on POSSE's gold set.

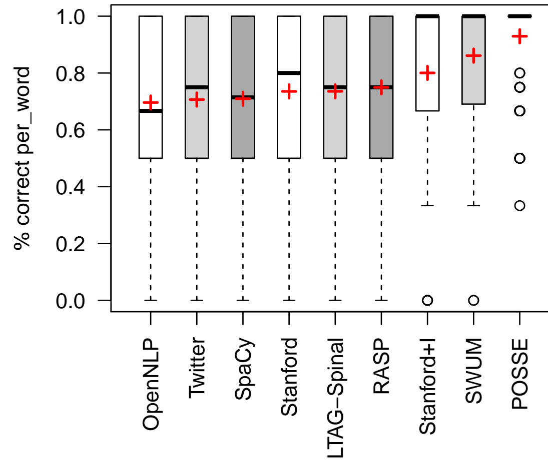

FIGURE 4.4.1: Mean word-level accuracy per identifier for POSSE's gold set

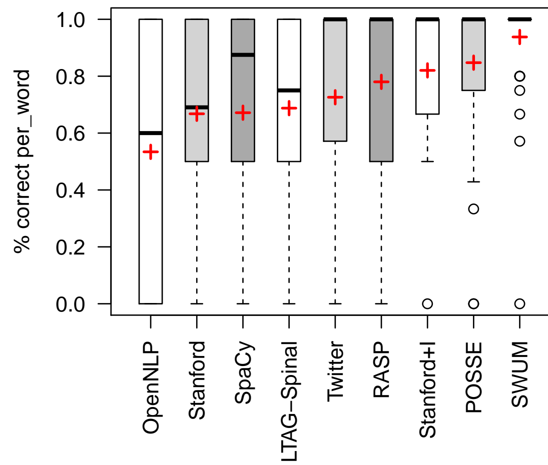

FIGURE 4.4.2: Mean word-level accuracy per identifier for supplemental gold set

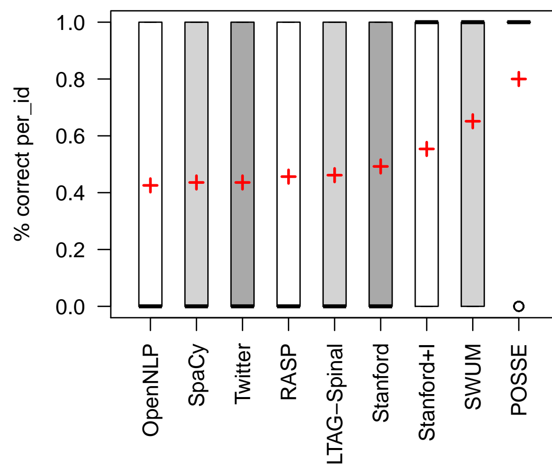

FIGURE 4.4.3: Mean identifier accuracy for POSSE's gold set

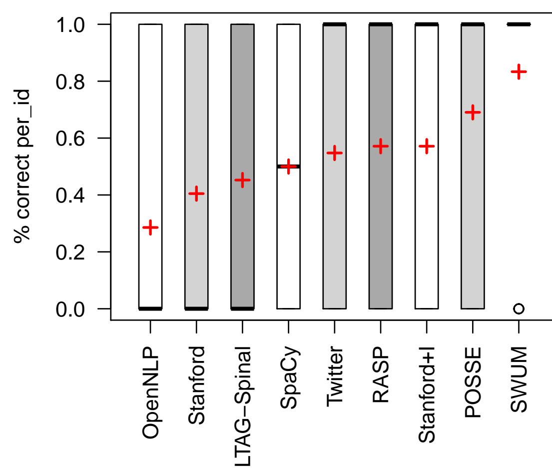

FIGURE 4.4.4: Mean identifier accuracy for supplemental gold set

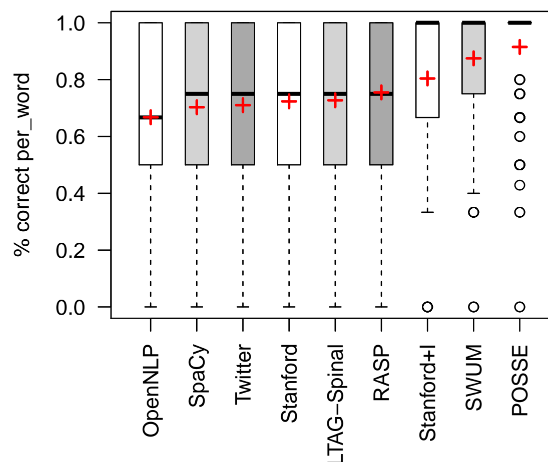

FIGURE 4.4.5: Overall per word accuracy across both sets

Additionally, figures 4.4.3 and 4.4.4 indicate the accuracy of taggers on phrases as a whole. For these figures, a tagger was evaluated to be accurate if it correctly tagged 100% of the words in a method name, and the scores given are the percent of the method names that were accurate. For the POSSE gold set, POSSE was once again the most accurate, followed by SWUM, and then "I" prepended Stanford. On the alternative gold set, POSSE also scored highest, followed closely by "I" prepended Stanford, and then by RASP.

As can be seen by looking at figures 4.4.1, 4.4.2, 4.4.3, and 4.4.4, prepending "I" to the method name appears to improve overall accuracy, compared to running the Stanford tagger on the same method names without being prepended. However, this difference is not statistically significant compared to the default behavior of the Stanford tagger.

In figure 4.4.5, the overall, per-word accuracy for each tagger can be seen, across

Taggers	Difference	Lower Bound	Upper Bound	P Score
Twitter-OpenNLP	0.0103540904	-0.077828427	0.09853661	0.9999913
SpaCy-OpenNLP	0.0132478632	-0.074934654	0.10143038	0.9999417
Stanford-OpenNLP	0.0390720391	-0.049110478	0.12725456	0.9068862
LTAG-Spinal-OpenNLP	0.0393284493	-0.048854068	0.12751097	0.9036242
RASP-OpenNLP	0.0532112332	-0.034971284	0.14139375	0.6319073
Stanford+I-OpenNLP	0.1041514042	0.015968887	0.19233392	0.0077278
SWUM-OpenNLP	0.1649816850	0.076799168	0.25316420	0.0000003
SpaCy-Twitter	0.0028937729	-0.085288744	0.09107629	1.0000000
Stanford-Twitter	0.0287179487	-0.059464568	0.11690047	0.9848829
LTAG-Spinal-Twitter	0.0289743590	-0.059208158	0.11715688	0.9839945
RASP-Twitter	0.0428571429	-0.045325374	0.13103966	0.8514344
Stanford+I-Twitter	0.0937973138	0.005614797	0.18197983	0.0271342
SWUM-Twitter	0.1546275946	0.066445078	0.24281011	0.0000021
Stanford-SpaCy	0.0258241758	-0.062358341	0.11400669	0.9924867
LTAG-Spinal-SpaCy	0.0260805861	-0.062101931	0.11426310	0.9919720
RASP-SpaCy	0.0399633700	-0.048219147	0.12814589	0.8952371
Stanford+I-SpaCy	0.0909035409	0.002721024	0.17908606	0.0374185
SWUM-SpaCy	0.1517338217	0.063551305	0.23991634	0.0000037
LTAG-Spinal-Stanford	0.0002564103	-0.087926107	0.08843893	1.0000000
RASP-Stanford	0.0141391941	-0.074043323	0.10232171	0.9999041
Stanford+I-Stanford	0.0650793651	-0.023103152	0.15326188	0.3474244
SWUM-Stanford	0.1259096459	0.037727129	0.21409216	0.0003352
RASP-LTAG-Spinal	0.0138827839	-0.074299733	0.10206530	0.9999166
Stanford+I-LTAG-Spinal	0.0648229548	-0.023359562	0.15300547	0.3529904
SWUM-LTAG-Spinal	0.1256532357	0.037470719	0.21383575	0.0003492
Stanford+I-RASP	0.0509401709	-0.037242346	0.13912269	0.6862889
SWUM-RASP	0.1117704518	0.023587935	0.19995297	0.0027732
SWUM-Stanford+I	0.0608302808	-0.027352236	0.14901280	0.4445575

Table 4.4.1: P-Scores and Bounds of Tukey range separation test on original data set, using per word accuracy

Taggers	Difference	Lower Bound	Upper Bound	P Score
Stanford-OpenNLP	0.133730159	-0.08888204	0.3563424	0.6318257
SpaCy-OpenNLP	0.137471655	-0.08514054	0.3600838	0.5954805
LTAG-Spinal-OpenNLP	0.153571429	-0.06904077	0.3761836	0.4392195
Twitter-OpenNLP	0.191836735	-0.03077546	0.4144489	0.1553494
RASP-OpenNLP	0.246031746	0.02341955	0.4686439	0.0180091
Stanford+I-OpenNLP	0.286281179	0.06366898	0.5088934	0.0023428
POSSE-OpenNLP	0.313435374	0.09082318	0.5360476	0.0004924
SWUM-OpenNLP	0.403741497	0.18112930	0.6263537	0.0000011
SpaCy-Stanford	0.003741497	-0.21887070	0.2263537	1.0000000
LTAG-Spinal-Stanford	0.019841270	-0.20277092	0.2424535	0.9999989
Twitter-Stanford	0.058106576	-0.16450562	0.2807188	0.9964081
RASP-Stanford	0.112301587	-0.11031061	0.3349138	0.8183825
Stanford+I-Stanford	0.152551020	-0.07006117	0.3751632	0.4488444
POSSE-Stanford	0.179705215	-0.04290698	0.4023174	0.2262901
SWUM-Stanford	0.270011338	0.04739914	0.4926235	0.0055664
LTAG-Spinal-SpaCy	0.016099773	-0.20651242	0.2387120	0.9999998
Twitter-SpaCy	0.054365079	-0.16824712	0.2769773	0.9977475
RASP-SpaCy	0.108560091	-0.11405210	0.3311723	0.8448913
Stanford+I-SpaCy	0.148809524	-0.07380267	0.3714217	0.4846367
POSSE-SpaCy	0.175963719	-0.04664848	0.3985759	0.2519635
SWUM-SpaCy	0.266269841	0.04365765	0.4888820	0.0067401
Twitter-LTAG-Spinal	0.038265306	-0.18434689	0.2608775	0.9998281
RASP-LTAG-Spinal	0.092460317	-0.13015188	0.3150725	0.9321501
Stanford+I-LTAG-Spinal	0.132709751	-0.08990244	0.3553219	0.6416390
POSSE-LTAG-Spinal	0.159863946	-0.06274825	0.3824761	0.3815449
SWUM-LTAG-Spinal	0.250170068	0.02755787	0.4727823	0.0148386
RASP-Twitter	0.054195011	-0.16841718	0.2768072	0.9977969
Stanford+I-Twitter	0.094444444	-0.12816775	0.3170566	0.9237793
POSSE-Twitter	0.121598639	-0.10101355	0.3442108	0.7436097
SWUM-Twitter	0.211904762	-0.01070743	0.4345170	0.0762841
Stanford+I-RASP	0.040249433	-0.18236276	0.2628616	0.9997488
POSSE-RASP	0.067403628	-0.15520857	0.2900158	0.9901686
SWUM-RASP	0.157709751	-0.06490244	0.3803219	0.4009319
POSSE-Stanford+I	0.027154195	-0.19545800	0.2497664	0.9999876
SWUM-Stanford+I	0.117460317	-0.10515188	0.3400725	0.7783433
SWUM-POSSE	0.090306122	-0.13230607	0.3129183	0.9405077

Table 4.4.2: P-Scores and Bounds of Tukey range separation test on supplemental data set, using per word accuracy

both of the data sets. By comparing this with figures 4.4.2 and 4.4.1, it can be seen that little changes when the two data sets are compared. SWUM and POSSE still rank highest, along with “Stanford+I”.

In table 4.4.2, the output of a Tukey Range Test can be seen. For $p<.05$, it can be seen that there is not a statistically significant difference between Spacy, Open-NLP, Twitter, Stanford, and LTAG-Spinal. RASP did perform significantly better than Open-NLP, but not more than any of the other taggers. The I-prepended Stanford tagger did not perform significantly better than the default case of Stanford. This implies that the strategy of prepending “I” to a method name would not significantly improve accuracy, but further study may be needed to confirm this across different taggers.

Finally, both POSSE and SWUM outperform every tagger to a statistically significant degree, except for Stanford+I. In the case of Stanford+I, POSSE significantly outperforms Stanford+I, but SWUM does not. Additionally, between POSSE and SWUM, there does not exist a statistically significant difference.

4.5 Discussion & Qualitative analysis

As can be seen from the preceding figures, SWUM and POSSE consistently had the highest accuracy across the two sets, while OpenNLP had the lowest. However, for most taggers, the change in accuracy was not large. It is worth noting that consistently one of the better taggers across the two sets was the Stanford tagger, with each method prepended with an “I”. In all cases, this approach outperformed

the default behavior of the Stanford tagger. For a detailed breakdown of the number and kinds of errors, see table 4.5.1. These tables present a summary of the types of mistakes each tagger made, and how frequently the mistakes were made.

The most common error was mislabeling a verb as a noun, which occurred 277 times across the two data sets. Additionally, nouns were mislabeled as verbs 148 times. These two mistakes are very similar. There are two types of this error which appear in the experiments that were run. The first is in the case of single word methods. An example of this is the method called exit. In normal language, this word can be either a verb or a noun. However, in source code, such a single word method name will generally be a verb, because it is an action to be taken. However, each tagger, except for RASP, SWUM and I-prepended Stanford, labeled it as a noun.

A second type of error which appeared frequently was when a phrase began with a word that could be either a verb or a noun. An example of this is configureWebConnection. This method begins with a verb, as "configure" is the main action being taken by the method. However, in this case, the only two taggers to get this correct were SWUM and I-prepended Stanford.

The opposite case, with a noun being mislabeled as a verb was less common, but still fairly frequent. This error occurred 148 times across all taggers. The tagger that most frequently made this mistake was the I-prepended Stanford. The purpose of prepending I to a method name was to try to force the tagger to recognize the first word as a verb by mimicking more traditional English grammatical structure, as a proper noun ("I") will usually be followed by a verb. The downside to this is that the assumption is made that the first word of the normal method is a verb. This

Error-> Expected Tag	POSSE	Stanford	Stanford+I	Twitter	RASP	OpenNLP	SpaCy	SWUM	LTAG-Spinal	Total
N->V	16	38	2	56	17	50	40	13	45	277
NP->N	0	30	29	28	25	30	31	0	33	206
V->N	1	21	36	13	26	12	10	20	9	148
PP->V	0	20	12	2	9	23	21	0	17	104
N->ADJ	22	7	4	9	4	8	6	30	7	97
ADJ->N	2	10	8	10	20	11	21	0	7	89
V3->V	0	11	11	12	10	10	11	0	10	75
ADJ->V	1	6	3	7	5	13	10	0	9	54
UN->N	0	5	5	3	11	0	2	2	0	28
PP->ADJ	1	2	3	4	5	3	3	0	3	24
V3->N	0	2	2	4	7	2	1	0	1	19
N->V3	4	1	1	2	1	1	1	5	2	18
N->PP	2	1	2	0	0	0	2	7	0	14
P->N	0	3	2	1	0	3	1	0	3	13
V->ADJ	0	0	2	0	3	0	0	6	0	11
P->V	1	1	1	1	1	1	1	1	1	9
VM->N	0	1	0	3	1	3	0	0	1	9
PP->N	0	1	1	1	2	1	2	0	1	9
N->P	5	0	0	0	0	0	0	3	0	8
N->NP	4	0	0	0	0	0	0	4	0	8
DT->N	0	0	0	0	0	1	2	1	2	6
V->PP	4	0	0	0	0	0	0	0	0	4
VM->ADJ	0	1	1	0	2	0	0	0	0	4
UN->V	0	0	0	4	0	0	0	0	0	4
PR->N	0	0	0	1	0	1	1	0	1	4
P->VM	0	1	0	1	1	0	0	0	0	3
VPR->VM	0	0	1	0	0	0	1	0	1	3
VPR->N	0	0	1	0	1	1	0	0	0	3
NP->V3	0	0	1	0	0	1	1	0	0	3
D->N	0	0	0	1	0	1	0	0	1	3
N->VM	1	0	0	0	0	0	0	1	0	2
VM->V	0	0	1	0	1	0	0	0	0	2
CJ->P	0	0	0	0	2	0	0	0	0	2
DT->ADJ	0	0	0	0	0	1	0	1	0	2
NP->V	0	0	0	0	0	1	0	0	1	2
NP->PP	0	0	0	0	0	2	0	0	0	2
UN->ADJ	0	0	0	0	0	0	0	2	0	2
N->DT	1	0	0	0	0	0	0	0	0	1
N->D	1	0	0	0	0	0	0	0	0	1
NP->ADJ	0	0	0	0	0	0	1	0	0	1
Total	66	162	129	163	154	180	169	96	155	

TABLE 4.5.1: Errors by tagger in both data sets

leads to increased accuracy overall, as I-prepended Stanford outperformed the default Stanford behavior across all experiments and measures, but it does carry a risk of creating new errors in the edge cases of methods beginning with nouns. An example of this risk is in the method called doubleArray, where double should be labeled as a noun, but I-prepended Stanford labeled it as a verb.

A second very common mistake was to mislabel a noun as a plural noun. While this is a mistake, and worth noting, it is unlikely to significantly impact the performance of an application designed to utilize POS information and process it. For example, in ReversedPreferencesMax, POSSE and SWUM both mislabeled Preferences as part of the noun phrase, but identified it as a singular noun, rather than a plural.

In table 4.5.1, several trends can be observed. First, despite the fact that they achieved the highest overall accuracy, SWUM and POSSE were the most likely to mistakenly label a noun as an adjective, one of the most common mistakes in the evaluation of the first data set. However, outside of this particular error, SWUM and POSSE had relatively few mistakes, while others had a number of other common mistakes. For example, every tagger other than SWUM and POSSE mistakenly labeled verbs as past-participles, a small but important distinction. In the new data set, the most common mistakes, were mislabeling verbs as nouns, and mislabeling adjectives as nouns.

In figure 4.5.1, a pattern of per-identifier accuracy can be seen. Each horizontal line represents an identifier. A blue line indicates that the tagger got this tag correct, while an orange line indicates that particular tagger got the tag wrong. This is useful as it allows us to see both the overall accuracy of each tagger, as well as the

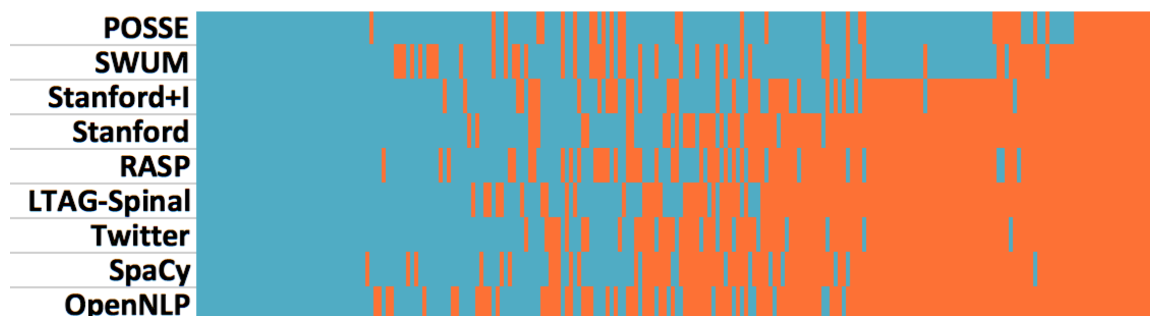

Figure 4.5.1: Color-coded accuracy table for different part-of-speech taggers

prominence of phrases that either no tagger got correct, or which only one tagger got correct. Additionally, it allowed us to begin to see patterns where different taggers complement each other, allowing for potentially greater accuracy.

By looking at figure 4.5.1, and table 4.5.1, it is possible to see why SWUM and POSSE outperformed the other taggers. In addition to having the highest accuracy on a per-identifier basis, (as established in figures 4.4.1 and 4.4.2), SWUM and POSSE were often able to avoid mistakes that were made by other taggers, such as the complete absence of verbs being labelled as past-participles, a common mistake in this experiment. The only types of errors that were more prominent in SWUM's and POSSE's outputs were mislabeling adjectives as nouns. Additionally, SWUM made only 96 mistakes across all tags, and POSSE made only 66 mistakes across all tags, significantly less than what were made by each other tagger. Interestingly, on the types of mistakes that SWUM and POSSE made relatively frequently, Stanford+I, a slightly less effective tagger, tended to avoid making mistakes. For example, Stanford+I mislabeled adjectives as nouns four times, tied for the least frequent occurrence of these mistakes with RASP. The only other frequent mistake for POSSE

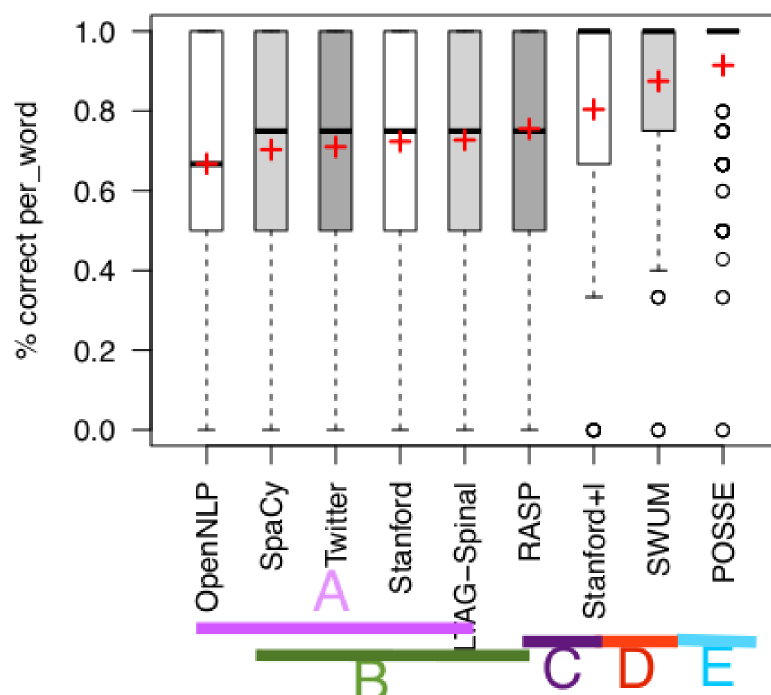

FIGURE 4.5.2: Groupings of taggers into equivalency groups

and SWUM was mislabelling verbs as nouns. Stanford+I made this mistake the least frequently, having only two instances of this mistake. This implies that it may be possible to combine these two taggers in a way that maximized the effectiveness of them both. For an illustration of the groupings of equivalent POS taggers, see 4.5.2. In this figure, the horizontal lines illustrate groups of taggers which do not differ statistically significantly from each other. For an illustration of the groupings of equivalent POS taggers, see 4.5.2. In this figure, the horizontal lines illustrate groups of taggers which do not differ statistically significantly from each other.

Chapter 5

Conclusions and Future Work

This thesis made a tool which could summarize statements of source code for a novice programmer. This tool, while functional, has several shortcomings as well. Most notably, Code Teacher, in its current state, only utilizes natural language for summarization of method calls and declarations. It may be possible to generate further, and more detailed summaries of source code artifacts by extending POS tagging to other types of artifacts in source code. Additionally, Code Teacher could be improved by pulling in information from other parts of the source code, such as comments and variable names.

While in chapter 3, this thesis demonstrated the possibility of summarizing source code without POS information, this can be improved with the inclusion of this type of information. For example, consider the summary of the methods `String getCompareString(Track track)` and `int compare(Object arg0, Object arg1)`. Without part of speech

information, the summary generated is "Declares a method to perform getCompare-String" and "Declares a method to perform compare", respectively. With accurate POS information, the generated summaries would resemble "Declares a method to get a compare string, using the Track provided", and "Declares a method to compare two objects", respectively.

Thus, this thesis also conducted a study of the effectiveness of part-of-speech taggers on the corpus of source code method names. This is novel research that included the process of unifying the tagsets of different part-of-speech taggers, a process which required studying and comparing divergent tagsets into a simple comparison. Additionally, part-of-speech tagging was conducted on a gold set of method names, and the accuracy of each tagger on this set was evaluated. Then, after this research was conducted, the usefulness of this research was demonstrated by developing a tool that could be used to generate summaries of source code for novice programmers, in a way that utilized part-of-speech information to make the summaries as detailed and accurate as possible.

In summary, this thesis made the following contributions:

- Conducted a comparative study of part-of-speech taggers on source code method names
- Developed a tool capable of parsing source code and generating explanatory statements of source code for novice programmers
- Illustrated the viability of using part-of-speech information in generating more detailed explanations of source code

This thesis provided evaluation of several part-of-speech taggers on the corpus of source code natural language artifacts. However, this was not an exhaustive study, and several taggers were not considered as part of the experiment. Further analysis of these taggers could be performed. Additionally, as illustrated by the “I-prepended” Stanford data set, it may be possible to improve upon the accuracy of taggers on the corpus of source code artifacts by mimicking natural language structure. Furthermore, further research could be conducted into studying what types of taggers make what kinds of mistakes more often, and if it would be possible to combine different taggers to make even more accurate results than either tool could achieve independently.

For the POS tagger implemented into Code Teacher, the decision was made to use SWUM. There were several reasons for this decision. SWUM was a high-performing and competitive tool, compared to other taggers. Additionally, for ease of implementation, SWUM provided high level method summaries without needing additional work. This will be implemented into “Code Teacher” as a future contribution.

Bibliography

[1] Surafel Lemma Abebe, Anita Alicante, Anna Corazza, and Paolo Tonella, *Supporting concept location through identifier parsing and ontology extraction*, J. Syst. Softw. **86** (2013), no. 11, 2919–2938.

[2] Surafel Lemma Abebe and Paolo Tonella, *Natural language parsing of program element names for concept extraction*, ICPC '10: Proceedings of the 2010 IEEE 18th International Conference on Program Comprehension (Washington, DC, USA), IEEE Computer Society, 2010, pp. 156–159.

[3] Reem S. AlSuhaibani, Christian D. Newman, Michael L. Collard, and Jonathan I. Maletic, *Heuristic-based part-of-speech tagging of source code identifiers and comments*, 2015 IEEE 5th Workshop on Mining Unstructured Data (2015), 1–6.

[4] Reem Saleh AlSuhaibani, *Part-of-speech tagging of source code identifiers using programming language context versus natural language context*, Master's thesis, Kent State University, 2015.

[5] Titus Barik, Jim Witschey, Brittany Johnson, and Emerson Murphy-Hill, *Compiler error notifications revisited: An interaction-first approach for helping developers more effectively comprehend and resolve error notifications*, Companion Proceedings of the 36th International Conference on Software Engineering (New York, NY, USA), ICSE Companion 2014, ACM, 2014, pp. 536–539.

[6] Dave Binkley, Matthew Hearn, and Dawn Lawrie, *Improving identifier informativeness using part of speech information*, Proceedings of the 8th Working Conference on Mining Software Repositories (New York, NY, USA), MSR '11, ACM, 2011, pp. 203–206.

[7] Ted Briscoe, John Carroll, and Rebecca Watson, *The second release of the rasp system*, Proceedings of the COLING/ACL on Interactive Presentation Sessions (Stroudsburg, PA, USA), COLING-ACL '06, Association for Computational Linguistics, 2006, pp. 77–80.

[8] Elizabeth Carter and Glenn D. Blank, *A tutoring system for debugging: Status report*, J. Comput. Sci. Coll. **28** (2013), no. 3, 46–52.

[9] Jinho D Choi, Joel Tetreault, and Amanda Stent, *It depends: Dependency parser comparison using a web-based evaluation tool.*

[10] Martha E Crosby, Jean Scholtz, and Susan Wiedenbeck, *The roles beacons play in comprehension for novice and expert programmers.*

[11] Leon Derczynski, Alan Ritter, Sam Clark, and Kalina Bontcheva, *Twitter part-of-speech tagging for all: Overcoming sparse and noisy data.*

[12] J.-R. Falleri, M. Huchard, M. Lafourcade, C. Nebut, V. Prince, and M. Dao, *Automatic extraction of a wordnet-like identifier network from software*, 18th Int'l Conf. on Program Comprehension, IEEE, jun. 2010, pp. 4 –13.

[13] Hiroyuki Fudaba, Yusuke Oda, Koichi Akabe, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura, *Pseudogen: A tool to automatically generate pseudo-code from source code*, (2015).

[14] Alex Gerdes, Johan Jeuring, and Bastiaan Heeren, *An interactive functional programming tutor*, Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education (New York, NY, USA), ITiCSE '12, ACM, 2012, pp. 250–255.

[15] S. Gupta, S. Malik, L. Pollock, and K. Vijay-Shanker, *Part-of-speech tagging of program identifiers for improved text-based software engineering tools*, Program Comprehension (ICPC), 2013 IEEE 21st International Conference on, 2013, pp. 3–12.

[16] Sonia Haiduc, Jairo Aponte, and Andrian Marcus, *Supporting program comprehension with source code summarization*, Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2 (New York, NY, USA), ICSE '10, ACM, 2010, pp. 223–226.

[17] Jiayun Han, *Free part-of-speech tagger.*

[18] ———, *Building an efficient, scalable, and trainable probability and rule based part-of-speech tagger of high accuracy*, (2009).

[19] Emily Hill, *A model of software word usage and its use in searching source code*, (2010).

[20] Emily Hill, Lori Pollock, and K. Vijay-Shanker, *Automatically capturing source code context of nl-queries for software maintenance and reuse*, Proceedings of the 31st International Conference on Software Engineering (Washington, DC, USA), ICSE '09, IEEE Computer Society, 2009, pp. 232–242.

[21] ———, *Improving source code search with natural language phrasal representations of method signatures*, Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering (Washington, DC, USA), ASE '11, IEEE Computer Society, 2011, pp. 524–527.

[22] Emily Hill, Shivani Rao, and Avinash C. Kak, *On the use of stemming for concern location and bug localization in java*, 12th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2012), IEEE Computer Society, 2012, pp. 184–193.

[23] https://github.com/ccrama/Slide, *Slide*.

[24] https://github.com/naman14/Timber, *Timber*.

[25] https://github.com/nickbutcher/plaid, *Plaid*.

[26] https://github.com/yahoo/anthelion, *Anthelion*.

[27] Andrew J. Ko and Brad A. Myers, *Extracting and answering why and why not questions about java program output*, ACM Trans. Softw. Eng. Methodol. **20** (2010), no. 2, 4:1–4:36.

[28] Ben Liblit, Andrew Begel, and Eve Sweetser, *Cognitive perspectives on the role of naming in computer programs*, 2006.

[29] Sana Malik, *Parsing java method names for improved software analysis*, Master's thesis, University of Delaware, 2011.

[30] Paul W. McBurney, Cheng Liu, Collin McMillan, and Tim Weninger, *Improving topic model source code summarization*, Proceedings of the 22Nd International Conference on Program Comprehension (New York, NY, USA), ICPC 2014, ACM, 2014, pp. 291–294.

[31] Brad A. Myers, David A. Weitzman, Andrew J. Ko, and Duen H. Chau, *Answering why and why not questions in user interfaces*, Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (New York, NY, USA), CHI '06, ACM, 2006, pp. 397–406.

[32] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura, *Learning to generate pseudo-code from source code using statistical machine translation*, (2015).

[33] Paige Rodeghero, Collin McMillan, Paul W. McBurney, Nigel Bosch, and Sidney D'Mello, *Improving automated source code summarization via an eye-tracking*

study of programmers, Proceedings of the 36th International Conference on Software Engineering (New York, NY, USA), ICSE 2014, ACM, 2014, pp. 390–401.

[34] Libin Shen, *Statistical ltag parsing*, Ph.D. thesis, Citeseer, 2006.

[35] Libin Shen and Aravind K. Joshi, *Incremental ltag parsing*, Proceedings of the Conference on Human Language Technology and Empirical Methods in Natural Language Processing (Stroudsburg, PA, USA), HLT '05, Association for Computational Linguistics, 2005, pp. 811–818.

[36] David Shepherd, Zachary P. Fry, Emily Hill, Lori Pollock, and K. Vijay-Shanker, *Using natural language program analysis to locate and understand action-oriented concerns*, AOSD '07: Proceedings of the 6th International Conference on Aspect-Oriented Software Development, 2007, pp. 212–224.

[37] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K. Vijay-Shanker, *Towards automatically generating summary comments for java methods*, Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (2010), 43–52.

[38] Giriprasad Sridhara, Emily Hill, Lori Pollock, and K Vijay-Shanker, *Identifying word relations in software: A comparative study of semantic similarity tools*, Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on, IEEE, 2008, pp. 123–132.

[39] Giriprasad Sridhara, Vibha Singhal Sinha, and Senthil Mani, *Naturalness of natural language artifacts in software*, Proceedings of the 8th India Software

Engineering Conference (New York, NY, USA), ISEC '15, ACM, 2015, pp. 156–165.

[40] Yuan Tian and David Lo, *A comparative study on the effectiveness of part-of-speech tagging techniques on bug reports*, SANER ERA, 2015, pp. 570–574.

[41] Kristina Toutanova, Dan Klein, Christopher D. Manning, and Yoram Singer, *Feature-rich part-of-speech tagging with a cyclic dependency network*, Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology - Volume 1 (Stroudsburg, PA, USA), NAACL '03, Association for Computational Linguistics, 2003, pp. 173–180.

[42] Yoshimasa Tsuruoka, Yusuke Miyao, and Jun'ichi Kazama, *Learning with lookahead: Can history-based models rival globally optimized models?*, Proceedings of the Fifteenth Conference on Computational Natural Language Learning (Stroudsburg, PA, USA), CoNLL '11, Association for Computational Linguistics, 2011, pp. 238–246.

[43] Yoshimasa Tsuruoka, Yuka Tateishi, Jin-Dong Kim, Tomoko Ohta, John McNaught, Sophia Ananiadou, and Jun'ichi Tsujii, *Developing a robust part-of-speech tagger for biomedical text*, Proceedings of the 10th Panhellenic Conference on Advances in Informatics (Berlin, Heidelberg), PCI'05, Springer-Verlag, 2005, pp. 382–392.

[44] Elaine Wong, Jinqiu Yang, and Lin Tan, *Autocomment: Mining question and answer sites for automatic comment generation*, Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on, IEEE, 2013, pp. 562–567.

[45] Annie T. T. Ying and Martin P. Robillard, *Selection and presentation practices for code example summarization*, Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (New York, NY, USA), FSE 2014, ACM, 2014, pp. 460–471.

[46] B. Zhang, E. Hill, and J. Clause, *Automatically generating test templates from test names (n)*, Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on, Nov 2015, pp. 506–511.

[47] Kurtis Zimmerman and Chandan R. Rupakheti, *An automated framework for recommending program elements to novices*, 2015 30th IEEE/ACM International Conference on Automated Software Engineering (2015), 283–288.

Appendix A

CodeTeacher Plug-in Source Code

In this appendix, snapshots are provided of the integration of new functionality into an Eclipse plugin. These are only the files modified as part of the research, where functionality was added.

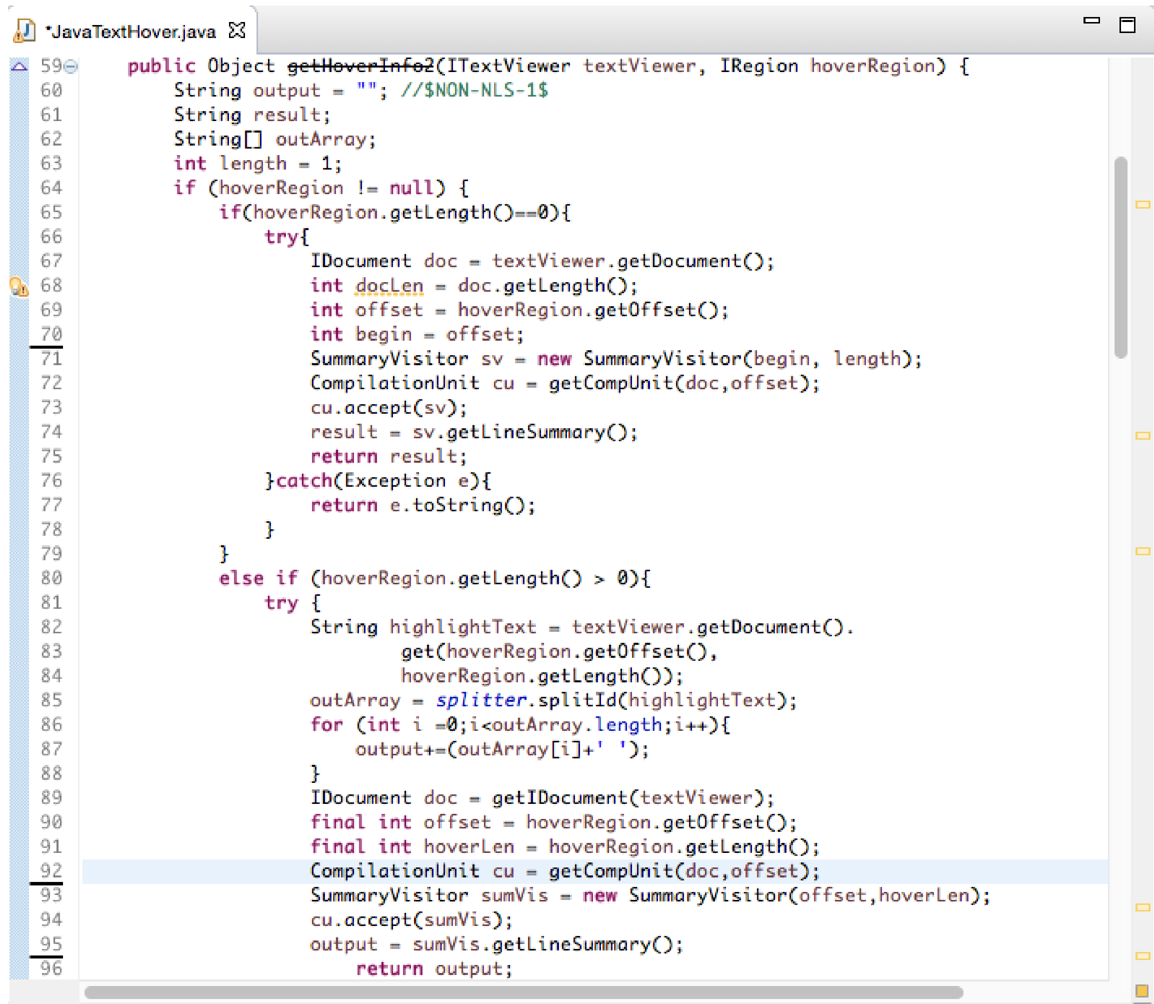

FIGURE A.0.1: Code responsible for hooking new SummaryVisitor class into existing framework (See https://goo.gl/gBy5vk for full code)

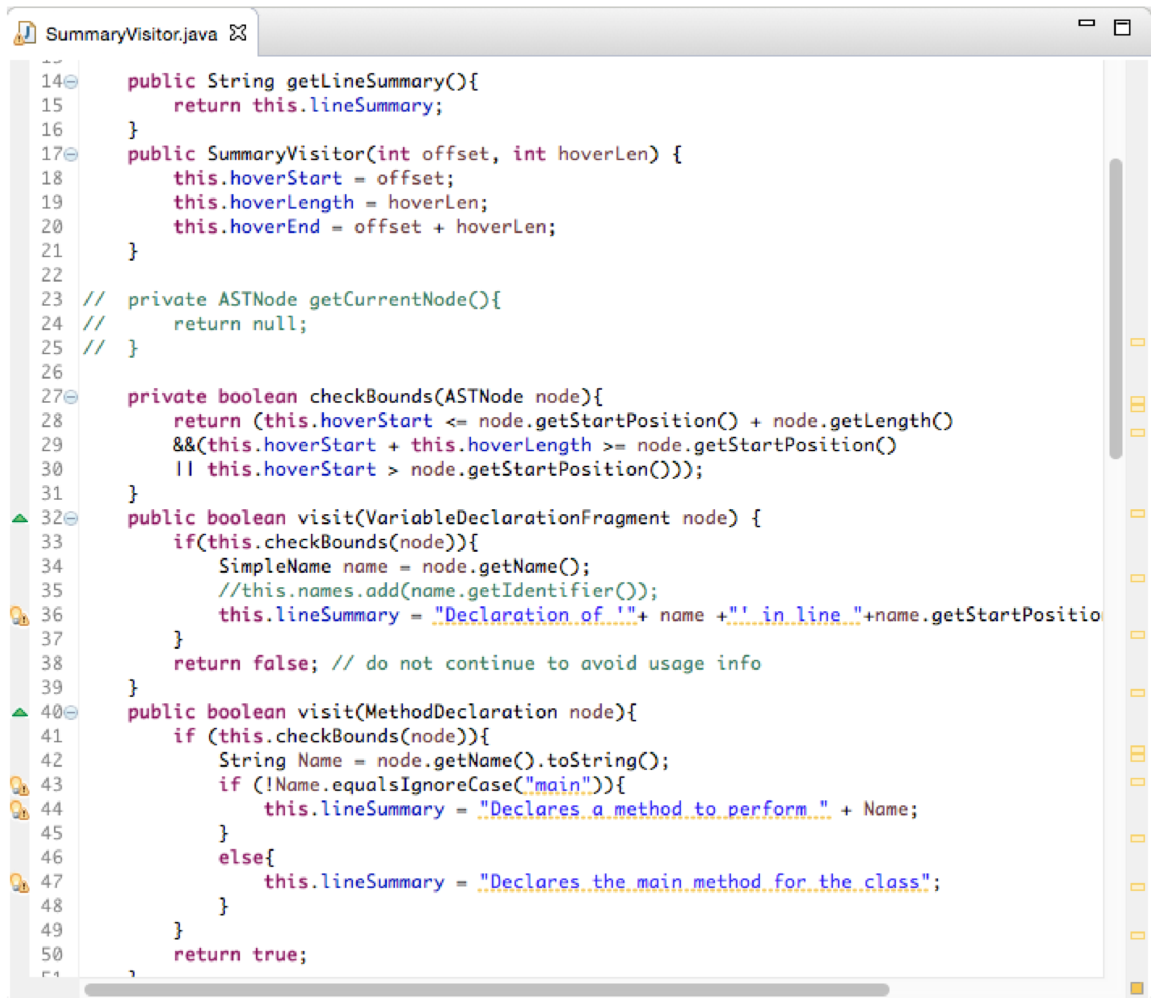

FIGURE A.0.2: Sample of code for new SummaryVisitor class, responsible for generating summaries and parsing AST (See https://goo.gl/WC3je1 for full code)

Appendix B

Gold Sets

See `goo.gl/OASpmn` and `goo.gl/FG26JN` for the data from the original and supplemental gold sets, including POS tags from each tagger. To evaluate accuracy, we took each phrase which we tagged manually, and converted it to its equivalent SWUM tag using table B.0.6. Each other tagset was also converted to their SWUM equivalency.

TABLE B.0.1: Original Gold Set with Tags

phrase	**POS tagged gold set**
get button layers	get-V button-N layers-NP
create using ctor	create-V Using-V3 Ctor-N
get single citation	get-V Single-ADJ Citation-N
round	round-V
experiment index exists	expiriment-N Index-N Exists-V
in progress	in-P Progress-N
is aiming at location	is-V Aiming-V3 At-P Location-N
is colonist	is-V Colonist-N
is consumer	is-V Consumer-N
is multiline	is-V Multiline-ADJ
is network enabled	is-V Network-N Enabled-VP
need swap saves	need-V Swap-VM Saves-NP
get instance	get-V Instance-N
double array	double-N Array-N
get orphans	get-V Orphans-NP
ordered map	ordered-AVP map-N
get major grid color	get-V major-ADJ grid-N color-N
get property pen color	get-V property-N pen-N color-N
get compiler env	get-V compiler-N env-N
action	action-N
cd scan view	cd-N scan-N view-N
color swatch	color-N swatch-N
confirm check box dialog	confirm-V check-N box-N dialog-N
drag tracker	drag-V tracker-N
hack	hack-N
help dialog	help-N dialog-N
move row down action	move-V row-N down-ADV action-N
move to front action	move-V to-P front-N action-N
nation	nation-N
out degree function	out-N degree-N function-N
report requirements action	report-VM requirements-NP action-N
smb connection	smb-N connection-N
start socket	start-V socket-N
tool error reporter	tool-N error-N reporter-N
torso twist action	torso-N twist-VM action-N
warehouse goods panel	warehouse-N goods-NP panel-N
sub	sub-N
update existing node	update-V existing-AV3 node-N
get display	get-V display-N
get std dev points visited	get-V std-N dev-N points-NP visited-VP
poisson	poisson-N
rev	rev-V
get exception	get-V exception-N
find source	find-V source-N
get next file	get-V next-ADJ file-N
get new project action	get-V new-ADJ project-N action-N
count colors	count-V colors-NP
get furniture icon step state	get-V furniture-N icon-N step-N state-N
place indian settlement	place-V indian-N settlement-N
arg max	arg-N max-N
bid	bid-V
compare to	compare-V to-P
convert from type	convert-V from-P type-N
first index of char	first-ADJ index-N of-P char-N
get cell status	get-V cell-N status-N
get f	get-V f-N
get index visible columns	get-V index-N visible-ADJ columns-NP
get maximum number of requests	get-V maximum-ADJ number-N of-P requests-NP

Table B.0.2: Original Gold Set with Tags

phrase	POS tagged gold set
get min value	get-V min-ADJ value-N
get m price list	get-V m-N price-N list-N
get price	get-V price-N
get sig data size	get-V sig-N data-N size-N
get status	get-V status-N
get tab index	get-V tab-N index-N
get usable width	get-V usable-ADJ width-N
get y	get-V y-N
get year	get-V year-N
index of outermost node	index-N of-P outermost-ADJ node-N
number of collection removals	number-N of-P collection-N removals-NP
remove nulls	remove-V nulls-NP
reversed preferences max	reversed-AVP preferences-NP max-N
get last	get-V last-N
create italic style toggle model	create-V italic-ADJ style-N toggle-N model-N
get tokens	get-V tokens-NP
get side pane plugin extension	get-V side-N pane-N plugin-N extension-N
get device stats	get-V device-N stats-NP
get munition type	get-V munition-N type-N
get L	get-V l-N
extract get method	extract-V get-VM method-N
get menu	get-V menu-N
search	search-V
get player base	get-V player-N base-N
get relative label point	get-V relative-ADJ label-N point-N
get print writer	get-V print-VM writer-N
get usable screen bounds	get-V usable-ADJ screen-N bounds-NP
get user obj for region	get-V user-N obj-N for-P region-N
get user obj for region	get-V user-N obj-N for-P region-N
calculate default download location	calculate-V default-N download-N location-N
get check select	get-V check-N select-N
acuity tip text	acuity-N tip-N text-N
get attach line	get-V attach-N line-N
get az style client name	get-v az-N style-N client-N name-N
get label	get-V label-N
get menu title	get-V menu-N title-N
get method fqn	get-V method-N fqn-N
get prefix	get-V prefix-N
get request key	get-V request-N key-N
get root table name	get-V root-N table-N name-N
get string for sentence	get-V string-N for-P sentence-N
insert record name	insert-V record-N name-N
literalize	literalize-V
show gui tip text	show-V gui-N tip-N text-N
show input dialog	show-V input-N dialog-N
new table	new-ADJ table-N
get step icon	get-V step-N icon-N
authenticate	authenticate-V
accept training set	accept-V training-AV3 set-N
activate components	activate-V components-NP
add	add-V
add deletion listener	add-V deletion-N listener-N
add fragment	add-V fragment-N
add handler	add-V handler-N
add new lines	add-V new-ADJ lines-NP
add option info	add-V option-N info-N
add	add-V
c label padding	c-N label-N padding-N
clear case delegate	clear-V case-N delegate-N

Table B.0.3: Original Gold Set with Tags

phrase	POS tagged gold set
close button action performed	close-VM button-N action-N performed-VP
close download bars	close-V download-N bars-N
close queue dispatch	close-V queue-N dispatch-N
compute third area	compute-V third-ADJ area-N
configure	configure-V
connect	connect-V
data source changed	data-N source-N changed-VP
dispose component	dispose-V component-N
dispose data source	dispose-V data-N source-N
do second pass	do-V second-ADJ pass-N
duration colors set enabled	duration-N colors-N set-V enabled-VP
end schema definition	end-V schema-N definition-N
engine init	engine-N init-N
engine update	engine-N update-V
error list pane	error-N list-N pane-N
find closest contacts	find-V closest-ADJ contacts-NP
fire activity started	fire-V activity-N started-VP
fire event	fire-V event-N
game board new	game-N board-N new-ADJ
improve solutions	improve-V solutions-NP
init root logger	init-V root-N logger-N
insert	insert-V
j menu item tile anondine action performed	j-N menu-N item-N tile-N anodine-N Action-N Performed-VP
j unit test listener	junit-N test-N listener-N
log j unit start	log-V junit-N start-N
log no new line	log-V no-DT new-ADJ line-N
make menu item	make-V menu-N item-N
make smb key	make-V smb-N Key-N
new task color option	new-ADJ task-N color-N option-N
parse char array	parse-V char-N array-N
parse filter def	parse-V filter-N def-N
popup help	popup-N help-N
prepare for phase	prepare-V for-P phase-N
process root return	process-V root-N return-N
process word	process-V word-N
refresh activity	refresh-V activity-N
remove property change listener	remove-V property-N change-N listener-N
remove	remove-V
remove walls	remove-V walls-NP
render	render-V
replace att	replace-V att-N
report progress	report-V progress-N
request write select	request-V write-VM select-N
restore item status	restore-V item-N status-N
safe j unit static inner class	safe-ADJ j-N unit-N static-ADJ inner-ADJ class-N
secure colony	secure-V colony-N
send have	send-V have-AV
set basic y pos	set-V basic-ADJ y-N pos-N
set bookmarks	set-V bookmarks-NP
set buddies	set-V buddies-NP
set collapsed paths	set-V collapsed-AVP paths-NP
set default batch fetch size	set-V default-ADJ batch-N fetch-VM size-N
set default font	set-V default-ADJ Font-N
set deployment complete	set-V deployment-N complete-ADJ

Table B.0.4: Original Gold Set with Tags

phrase	POS tagged gold set
set file	set-V file-N
set method to call	set-V method-N to-INF call-V
set num folders mi option	set-V num-N folders-NP mi-N option-N
set pie dataset	set-V pie-N dataset-N
set q name	set-V q-N name-N
set resource color	set-V resource-N color-N
set shape	set-V shape-N
set split point	set-V split-N point-N
set spotlight state	set-V spotlight-N state-N
set stops	set-V stops-NP
set style	set-V style-N
set tc mapping	set-V tc-N mapping-N
set transform	set-V transform-N
set x	set-V x-N
test clear current	test-V clear-V current-N
test indent common cases	test-V indent-V common-ADJ cases-NP
tree nodes changed	tree-N nodes-NP changed-VP
update font	update-V font-N
update preview	update-V preview-N
update	update-V
update source image	update-V source-N image-N
visit ancestors	visit-V ancestors-NP
create cllrm15	create-V CLLRM15-N
create isherppc	create-V ISHERPPC-N
create word	create-V word-N

Table B.0.5: Supplemental Gold Set with Tags

phrase	**POS tagged gold set**
populate	populate-V
populate Tree	populate-V tree-N
populate Tree By Style	populate-V tree-N by-P style-N
run	run-V
equals	equals-V
get Name 2	get-V name-N 2-#
get Style	get-V style-N
compare	compare-V
get Compare String	get-v compare-VM string-n
Track Comparator	Track-N Comparator-N
get Comparator	get-V comparator-N
Track Manager	track-N manager-N
set Comparator	set-V Comparator-N
I Report Compiler	I-N Report-N Compiler-N
run	run-V
start	start-V
get Translated Compile Directory	get-V Translated-VM Compile-VM Directory-N
is Using Current Files Directory For Compiles	is-V Using-V Current-ADJ Files-N Directory-N For-P Compiles-V3
j Button Compiler Action Performed	j-ADJ Button-N Compiler-N Action-N Performed-VP
j Button Run 1 Action Performed	j-ADJ Button-N Run-V 1-# Action-N Performed-VP
drop New Text Field	drop-V New-ADJ Text-N Field-N
drop	drop-V
Text Field Report Element	Text-N Field-N Report-N Element-N
Text Report Element	Text-N Report-N Element-N
Search Result	Search-N Result-N
to String Search	to-P String-N Search-N
search	search-V
get Resu	get-V Resu-N
reload	reload-V
Configure Web Connection	Configure-V Web-N Connection-N
start	start-V
Movie Item	Movie-N Item-N
find Shows	find-V Shows-NP
Compiler Environs	Compiler-N Environs-N
is Reserved Keyword As Identifier	is-V Reserved-ADJ Keyword-N as-P Identifier-N
set Reserved Keyword As Identifier	set-V Reserved-ADJ Keyword-N As-P Identifier-N
call	call-V
enter	enter-V
exit	exit-V
has Feature	has-V Feature-N
decompile	decompile-V

Table B.0.6: Conversion Table for Gold to SWUM tagsets

Tag	**SWUM Equivalent**
Adjectival Verb Past tense (AVP)	ADJ
Conjunction (CJ)	CJ
Digit (#)	D
Determiner (DT)	DT
Noun (N)	N
Adjectival Noun (AN)	N
Pronoun (PR)	N
Adjective (ADJ)	NM
Plural Noun (NP)	NP
Adjectival Plural Noun (ANP)	NP
Preposition (P)	P
Past tense verb (VP)	PP
Unknown (UN)	UN
Abreviation (ABV)	UN
Verb (V)	V
Adjectival Verb (AV)	V
Infinitive (to)	V
Third Person Verb (V3)	V3
Adjectival third person verb (AV3)	V3
Adverb (ADV)	VM