On the Generation, Structure, and Semantics of Grammar Patterns in Source Code Identifiers

Christian D. Newman[1,*], Reem S. AlSuhaibani[2], Michael J. Decker[3], Anthony Peruma[1], Dishant Kaushik[1], Mohamed Wiem Mkaouer[1], Emily Hill[4]

Abstract

Identifiers make up a majority of the text in code. They are one of the most basic mediums through which developers describe the code they create and understand the code that others create. Therefore, understanding the patterns latent in identifier naming practices and how accurately we are able to automatically model these patterns is vital if researchers are to support developers and automated analysis approaches in comprehending and creating identifiers correctly and optimally. This paper investigates identifiers by studying sequences of part-of-speech annotations, referred to as grammar patterns. This work advances our understanding of these patterns and our ability to model them by 1) establishing common naming patterns in different types of identifiers, such as class and attribute names; 2) analyzing how different patterns influence comprehension; and 3) studying the accuracy of state-of-the-art techniques for part-of-speech annotations, which are vital in automatically modeling identifier naming patterns, in order to establish their limits and paths toward improvement. To do this, we manually annotate a dataset of 1,335 identifiers from 20 open-source systems and use this dataset to study naming patterns, semantics, and tagger

[*] Corresponding author

Email addresses: cnewman@se.rit.edu (Christian D. Newman), ralsuhai@kent.edu (Reem S. AlSuhaibani), mdecke@bgsu.edu (Michael J. Decker), axp6201@rit.edu (Anthony Peruma), dkaushik95@gmail.com (Dishant Kaushik), mwmvse@rit.edu (Mohamed Wiem Mkaouer), emhill@drew.edu (Emily Hill)

[1] Rochester Institute of Technology, Rochester, NY, USA

[2] Kent State University, Kent, OH, USA, Prince Sultan University, Riyadh, KSA

[3] Bowling Green State University, Bowling Green, OH, USA

[4] Drew University, Madison, NJ, USA

accuracy.

Keywords: Program Comprehension, Identifier Naming, Software Maintenance, Source Code Analysis, Part-of-speech Tagging

1. Introduction

Currently, developers spend a significant amount of time comprehending code [1, 2]; 10 times the amount they spend writing it by some estimates [2]. Studying comprehension will lead to ways that not only increase the ability of developers to be productive, but also increase the accessibility of software development (e.g., by supporting programmers that prefer top-down or bottom-up comprehension styles [3, 4]) as a field. One of the primary ways a developer comprehends code is by reading identifier names which make up, on average, about 70% of the characters found in a body of code [5].

Recent studies show how identifier names impact comprehension [6, 7, 8, 9, 10], others show that normalizing identifier names helps both developers and research tools which leverage identifiers [11, 12]. Thus, many research projects try to improve identifier naming practices. For example, prior research predicts words that should appear in an identifier name [13, 14, 15, 16]; combines Natural Language (NL) and static analysis to analyze or suggest changes to identifiers [17, 18, 19, 20, 16, 21, 22, 23, 24]; and groups identifier names by static role in the code [25, 26, 27].

One challenge to studying identifiers is the difficulty in understanding how to map the meaning of natural language phrases to the behavior of the code. For example, when a developer names a method, the name should reflect the behavior of the method such that another developer can understand what the method does without the need to read the method body instruction set. Understanding this connection between name and behavior presents challenges for humans and tools; both of which use this relationship to comprehend, generate, or critique code. A second challenge lies in the natural language analysis techniques themselves, many of which are not trained to be applied to software

[11], which introduces significant threats [28]. Addressing these problems is vital to improving the developer experience and augmenting tools which leverage natural language.

One of the most popular methods for measuring the natural language semantics of identifier names is part-of-speech (POS) tagging. While some work has been done studying part-of-speech tag sequences (i.e., grammar patterns) in identifier names [20, 23, 29, 30, 31, 32], these prior works either focus on a specific type of identifier, typically method or class names; or discuss grammar patterns at a conceptual level without showing concrete examples of what they look like in the wild, where they can be messy, incomplete, ambiguous, or provide unique insights about how developers express themselves. In this paper, we begin addressing these issues. We create a dataset of 1,335 manually-annotated (i.e., POS tagged) identifiers across five identifier categories: class, function, declaration-statement (i.e., global or function-local variables), parameter, and attribute names. We use this dataset to study and show concrete grammar patterns as they are found in their natural environments.

The goal of this paper is to study the structure, semantics, diversity, and generation of grammar patterns. We accomplish this by 1) establishing and exploring the common and diverse grammar pattern structures found in identifiers. 2) Using these structures to investigate the accuracy, strengths, and weaknesses of approaches which generate grammar patterns with an eye toward establishing and improving their current ability. Finally, 3) leveraging the grammar patterns we discover to discuss the ways in which words, as part of a larger identifier, work together to convey information to the developer. We answer the following Research Questions (RQs):

RQ1: What are the most frequent human-annotated grammar patterns and what are the semantics of these patterns? This question explores the top 5 frequent patterns generated by the human annotators and discusses what bearing these patterns have on comprehending the connection between identifiers and code semantics/behavior.

RQ2: How accurately do the chosen taggers annotate grammar

patterns and individual tags? This question explores the accuracy of the part-of-speech tagger annotations versus human annotations. We determine which patterns and part-of-speech tags are most often incorrectly generated by tools.

RQ3: Are there other grammar patterns that are dissimilar from the most frequent in our data, but still present in multiple systems? This question explores patterns which are not as frequent as those discussed in RQ1. We manually pick a set of patterns that are structurally dissimilar from the top 5 from RQ1, but still appear in multiple systems within the dataset. Consequently, we identify unique patterns which are not as regularly observed as our top 5 patterns, but are repeatedly represented in the dataset and important to discuss. This question addresses the diversity of patterns within the dataset.

RQ4: Do grammar patterns or tagger accuracy differ across programming languages? This question explores how grammar patterns compare when separated by programming language. We determine how C/C++ and Java grammar patterns are structurally similar and dissimilar from one another. We also analyze tagger accuracy when we split grammar patterns by programming language.

We also discuss the ways which we will use this data in our future work. The results of this study:

- Increase our understanding of how different grammar patterns convey information.
- Provide a list of patterns which may be used to both understand naming practices and identify further patterns requiring further study.
- Highlight the accuracy of, and ways to improve, part-of-speech taggers.
- Highlight the prevalence, and usage, of part-of-speech annotations which are found in source code.

This paper is organized as follows, Section 2 gives the necessary background related to grammar pattern generation. Our methodology is detailed in Section

3. Experiments, driven by answering our research questions, are in Section 4. Section 5 enumerates all the threats to the validity of our experiments. Section 6 discusses studies related to our work, before further discussing our findings in Section 7 and concluding in Section 8.

2. Definitions & Grammar Pattern Generation

In this paper, we use grammar patterns to understand the relationship between groups of identifiers. A *grammar pattern* is the sequence of part-of-speech tags (also referred to as annotations) assigned to individual words within an identifier. For example, for an identifier called GetUserToken, we assign a grammar pattern by splitting the identifier into its three constituent words: Get, User, and Token. We then run the split-sequence (i.e., Get User Token) through a part-of-speech tagger to get the grammar pattern: Verb Adjective Noun. Notice this grammar pattern is not unique to this identifier, but is shared with many potential identifiers that use similar words. Thus, we can relate identifiers that contain different words to one another using their grammar pattern; GetUserToken, RunUserQuery, and WriteAccessToken share the same grammar pattern and, while they do not express the exact same semantics, there are similarities in their semantics which their grammar patterns reveal. Specifically, a verb (get, run, write) applied to a noun (token, query) with a specific role/context (user, access).

2.1. Generating Grammar Patterns

To generate grammar patterns for identifiers in source code, we require a part-of-speech tagger and a way to split identifiers into their constituent terms. To split terms in an identifier, we use Spiral [33]; an open-source tool combining several splitting approaches into one Python package. From this package, we used heuristic splitting and manually corrected mistakes made by the splitter in our manually-annotated set, which we discuss in the next section. We generate grammar patterns by running each individual tagger on each identifier

Table 1: Examples of grammar patterns

Identifier Example	**Grammar Pattern**
1. GList* **tile list head** = NULL;	adjective adjective noun
2. GList* **tile list tail** = NULL;	adjective adjective noun
3. Gulong **max tile size** = 0;	adjective adjective noun
4. GimpWireMessage **msg**;	noun
5. **g list remove link** (tile list head, list)	preamble noun verb noun
6. **g list last** (list)	preamble adjective noun
7. **g assert** (tile_list_head != tile_list_tail);	preamble verb

after applying our splitting function. Additionally, we give examples of grammar patterns in Table 1, which shows a set of identifiers on the left and the corresponding grammar pattern on the right.

Since part of the goal of this paper is to study POS taggers, we use multiple taggers on our identifier dataset; Posse [23], Swum [34], and Stanford [35] [5]. Posse and Swum are part-of-speech taggers created specifically to be run on software identifiers; they are trained to deal with the specialized context in which identifiers appear. Both Posse and Swum take advantage of static analysis to provide annotations. For example, they will look at the return type of a function to determine whether the word *set* is a noun or a verb. Additionally, they are both aware of common naming structures in identifier names. For example, methods are more likely to contain a verb in certain positions within their name (e.g., at the beginning) [23, 34]. They leverage this information to help determine what POS to assign different words. Stanford is a popular POS tagger for general natural language (e.g., English) text. For our study, Stanford provides a baseline; it is not specialized for source code but is reasonably accurate on method names [36].

[5] Version: 3.9.2, taggermodel: english-bidirectional-distsim.tagger, jdk version: openJDK 11.0.7

Table 2: Part-of-speech categories used in study

Abbreviation	**Expanded Form**	**Examples**
N	noun	Disneyland, shoe, faucet, mother, bedroom
DT	determiner	the, this, that, these, those, which
CJ	conjunction	and, for, nor, but, or, yet, so
P	preposition	behind, in front of, at, under, beside, above, beneath, despite
NPL	noun plural	Streets, cities, cars, people, lists, items, elements.
NM	noun modifier	red, cold, hot, scary, beautiful, happy, faster, small
V	verb	Run, jump, drive, spin
VM	verb modifier (adverb)	Very, loudly, seriously, impatiently, badly
PR	pronoun	she, he, her, him, it, we, us, they, them, I, me, you
D	digit	1, 2, 10, 4.12, 0xAF
PRE	preamble*	Gimp, GLEW, GL, G

This paper uses the part-of-speech tags given in Table 2. Note, in this table, the *preamble* category, which does not exist in general natural language part-of-speech tagging approaches. A Preamble is an abbreviation which does one of the following:

1. Namespaces an identifier without augmenting the reader's understanding of its behavior (e.g., XML in XML_Reader is not a preamble)
2. Provides language-specific metadata about an identifier (e.g., identifies pointers or member variables)
3. Highlights an identifier's type. When a preamble is highlighting an identifier's type, the type's inclusion must not add any new information to the identifier name.

For example, given an identifier *float* fPtr*, 'f' does not add any information about the identifier's role within the system, but reminds the developer that it has a type 'float'. However, given an identifier *char* sPtr*, 's' informs the developer that this is a c-string as opposed to a pointer to some other type of character array; 's' would not be considered a preamble under this definition. Additionally, some developers will put *p_* in front of pointer variables or m_ in front of variables that are members of a class; these are due to naming conventions and/or Hungarian notation [9, 31, 37, 38]. In the GIMP and GLEW open-source projects, GIMP and G_ are preambles to many variables, as seen in the Gimp example in Table 1. Intuitively, the reason for identifying preambles in an identifier is because they do not provide any information with respect to the identifier's role within the system's domain. Instead, they provide one of the three types of information above. For this reason, when analyzing identifiers, tools should be able to determine when a word is a preamble so that they do not make false assumptions about the word's descriptive purpose.

Another tag to note from Table 2 is *noun modifier (NM)*, which is annotated on words that could be considered either pure adjectives or noun-adjuncts. A noun-adjunct is a word that is typically a noun but is being used as an adjective. An example of this is the word *bit* in the identifier *bitSet*. In this case, *bit* is a noun which describes the type of *set*, i.e., it is a set of bits. So we consider it a noun-adjunct. These are found in English (e.g., compound words), but generally not annotated as their own individual part-of-speech tag. In this work, we argue for the use of an individual tag for noun-adjuncts due to their ubiquity, and special role, in source code identifiers, which we discuss.

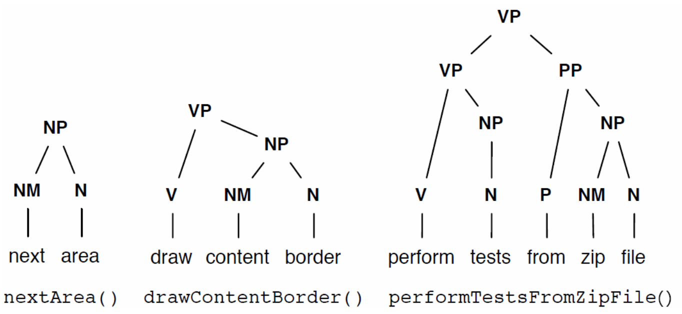

Figure 1: Examples of noun, verb, and prepositional phrases

The tagset in Table 2 is a smaller set than some standard natural language part-of-speech tagsets, such as the Penn Treebank tagset used by Stanford [35], due to the fact that Posse [23] and Swum [34] use this limited set. Because Swum/Posse both rely on a more limited set, we use a manually curated mapping from the Penn Treebank set to this narrower tagset, provided in Table 3, which we now discuss. Many of these mappings take subcategories of various part-of-speech annotations and translate them to the broadest category. For example, proper noun ⟶ noun and modal ⟶ verb, where a modal is just a more specific kind of verb, and a proper noun is a more specific kind of noun.

Past tense verb (VBD), present participle verb (VBG), and past participle verb (VBN) are all used as adjectives in many situations within our data set. For example, *sortedIndicesBuf*, *waitingList*, and *adjustedGradient* where *sorted* is a past tense verb (VBD), *waiting* is a present participle (VBG), and *adjusted* is a past participle verb (VBN). All three of these verbs can be used as adjectives. Therefore, we process these twice for Stanford, which is the only tagger that detects these verb subcategories: once as verbs and once as adjectives to compare Stanford's accuracy under both configurations. VBZ (third-person verbs) are also sometimes used as adjectives, but based on our data this annotation is

Table 3: How Penn Treebank annotations were mapped to the reduced set of annotations

Penn Treebank Annotation	Annotation Used In Study
Conjunction (CC)	Conjunction (CJ)
Digit (CD)	Digit (D)
Determiner (DT)	Determiner (DT)
Foreign Word (FW)	Noun (N)
Preposition (IN)	Preposition (P)
Adjective (JJ)	Noun Modifier (NM)
Comparative Adjective (JJR)	Noun Modifier (NM)
Superlative Adjective (JJS)	Noun Modifier (NM)
List Item (LS)	Noun (N)
Modal (MD)	Verb (V)
Noun Singular (NN)	Noun (N)
Proper Noun (NNP)	Noun (N)
Proper Noun Plural (NNPS)	Noun Plural (NPL)
Noun Plural (NNS)	Noun Plural (NPL)
Personal Pronoun (PRP)	Pronoun (PR)
Possessive Pronoun (PRP$)	Pronoun (PR)
Adverb (RB)	Verb Modifier (VM)
Comparative Adverb (RBR)	Verb Modifier (VM)
Particle (RP)	Verb Modifier (VM)
Symbol (SYM)	Noun (N)
To Preposition (TO)	Preposition (P)
Verb (VB)	Verb (V)
Verb (VBD)	Verb or Noun Modifier (V or NM**)
Verb (VBG)	Verb or Noun Modifier (V or NM**)
Verb (VBN)	Verb or Noun Modifier (V or NM**)
Verb (VBP)	Verb (V)
Verb (VBZ)	Verb (V)

most frequently used for the words *is* or *equals*; typically in boolean identifiers or methods. Therefore, we treat VBZ as a verb since treating them as an adjective would generally result in lower accuracy. The same applies to VB and VBP; words which receive these annotations are typically verbs being used as verbs in our dataset. Next, when Stanford assigns List Item (LS) and Symbol (SYM) to words in our data set, it is typically mis-annotating nouns, so we map these to a noun.

Finally, when we apply the Stanford tagger to function names, we append the letter *I* to the beginning of the name. This is a known technique– the Stanford+I technique, used to help Stanford tag identifiers that represent actions more accurately. It was used in previous studies applying part-of-speech tags to method identifier names [36] to increase Stanford's accuracy. We also test to make certain that this did, in fact, improve Stanford's output and present these results later in Section 4.2. Other research has augmented input to Stanford or other NLP techniques improve analysis in the past [22, 39].

2.2. Noun, Verb, and Prepositional phrases

There are a few linguistic concepts that come up when we analyze part-of-speech tagger output. Specifically, we will be dealing with noun phrases, verb phrases, and prepositional phrases. We define these terms and give an example. A Noun Phrase (NP) is a sequence of noun modifiers, such as noun-adjuncts and adjectives, followed by a noun, and optionally followed by other modifiers or prepositional phrases [40]. The noun in a noun phrase is typically referred to as a *head-noun*; the entity which is being modified/described by the words to its left [5] (or, for programmers, sometimes surrounding it) in the phrase. A Verb Phrase (VP) is a verb followed by an NP and optionally a Prepositional Phrase (PP). A PP is a preposition plus an NP and can be part of a VP or NP.

Figure 1 presents an example NP, VP, and VP with PP for three method name identifiers. The phrase structure nodes are NP, VP, and PP, while the other nodes (i.e., N, NM, V, P) are part-of-speech annotations. The leaf nodes are the individual words split from within the identifier. Each word in the identi-

Table 4: Total number per category of identifiers and unique grammar patterns across all systems

Category	Total Identifiers Across All Systems	# of Unique Grammar Patterns in Dataset
Decls	920778	45
Classes	37117	40
Functions	428748	96
Parameters	1197047	40
Attributes	159562	53
Total	2743252	277

fier is assigned a part-of-speech, which can then be used to derive the identifier's phrase structure. We cannot build a phrase structure without part-of-speech information. One important thing to note about these phrases is how the words in the phrases work together. For example, in noun phrases, noun modifiers (e.g., other nouns or adjectives) work to modify (i.e., specify) the concept represented by the head-noun that is part of the same phrase. In Figure 1, *contentBorder* is a noun phrase where *content* modifies our understanding of the noun *border*. It tells us that we are talking about the content border as opposed to another type of border; a *window border*, for example. When we make it into a verb phrase by adding draw to get *drawContentBorder*; we add an action (i.e., draw) that will be applied to the particular type of border (i.e., the content border) represented by the identifier.

3. Methodology

Identifiers come in many forms. The most common fall into one of the five following categories: class names, function names, parameter names, attribute names (i.e., data members), and declaration-statement names. A declaration-statement name is a name belonging to a function-local or global variable. We use this terminology as it is consistent with srcML's terminology [41] for these variables and we used srcML to collect identifiers. Therefore, to study grammar patterns, we group a set of identifiers based on which of these five categories they fell into. The purpose of doing this is two-fold. 1) We can study grammar pattern frequencies based on their high-level semantic role (i.e., class names have a different role than function names). 2) We can consider the differences in accuracy for part-of-speech taggers when given identifiers from different categories.

3.1. Setup for the dataset of human-annotated identifiers

We created a gold set of grammar patterns for each of the five categories above by manually assigning (i.e., annotating) part-of-speech tags to each word within each identifier within each of the five categories above. We calculate the sample size by counting the total number of identifiers in each of the five categories (given in Table 4) and calculating a sample based on a 95% confidence level and a confidence interval of 6%. We chose this confidence level and interval as a trade-off between time (i.e., annotating and validating is a manual effort) and accuracy. Using this confidence level and interval, we determine that each of our five categories needs to contain 267 samples (i.e., 267 was the largest number any of the sets required to be statistically representative, some required less– we went with 267). This totals to 1335 identifiers in the entire set. This sample size is also similar to the number used in prior studies on part-of-speech tagging [36, 23].

Initially, one author (annotator) was assigned to each category and was responsible for assigning grammar patterns for each of the 267 identifiers in the category. The annotators were given a split identifier (using Spiral [33]) along with the identifier's type and, if the identifier represented a function, the parameters/return types for that function. They were also allowed to look at the source code from which the identifier originated if needed. The annotators were asked to additionally identify and correct mistakes made by Spiral.

We did not expand abbreviations. The reason for this is that abbreviation expansion techniques are not widely available (e.g., cannot be easily integrated into different languages or frameworks, cannot be readily trained, are not fully or publicly implemented) and still not very accurate [12]. Therefore, a realistic worst-case scenario for developers and researchers is that no abbreviation-expansion technique available to use; their part-of-speech taggers must work in this worst-case scenario. We also tried not to split domain-term abbreviations (e.g., Spiral will make IPV4 into IPV 4; we corrected this back to IPV4). We did this because some taggers may recognize these domain terms. It is also the view of the authors that they should be recognized and appropriately tagged in their abbreviated (i.e., their most common) form. In the future, we plan to train a part-of-speech tagger using this dataset.

After completing their individual sets, the authors traded and reviewed one another's sets (i.e., performed cross-validation) twice. Thus, every identifier has been reviewed by two annotators. There was only one disagreement that could not be settled due to a particularly disfigured identifier; therefore, one identifier was randomly re-selected. This identifier is as follows: *uint8x16_t a_p1_q1_2*; the annotators could not ascertain the meaning of the letters and numbers, making it difficult to tag. Once every identifier was assigned a grammar pattern manually and had been reviewed by at least two other authors, we ran each of our three part-of-speech taggers on the set of split identifiers; providing whatever information was required by the tagger (e.g., some taggers require full function signature, others only use the identifier name). We used srcML [41] to obtain any additional information required by the taggers. The grammar pattern output from each tagger was used to generate frequency counts and compare to the manually-annotated grammar patterns to calculate accuracy.

3.2. Definition of Accuracy

Accuracy in this paper is synonymous with agreement; we compare the automatically generated annotations from the individual part-of-speech taggers with the manual annotations provided by humans. To be specific, we perform two different accuracy calculations for each tagger. One to determine the accuracy of each tagger on the individual part-of-speech tags found in Table 10 and one to determine the accuracy of each tagger on full grammar patterns like those in Table 6. To put this into an equation, we first define four sets. H_{gp}, the set of all human-annotated grammar patterns. T_{gp}, the set of all tool-annotated grammar patterns for a single part-of-speech tagger; there are three of these since we use three tools in this paper. H_{word}, the set of all human annotations for individual words in our set. Finally, T_{word}, the set of all tool annotations for individual words. Again, there are three of these since we use three part-of-speech tools in this paper. We then define grammar pattern level accuracy as the number of patterns which the human and tool sets agreed on (i.e., intersection) divided by the total number of grammar patterns annotated by humans. The equation follows:

$$\left|H_{gp} \cap T_{gp}\right| \div \left|H_{gp}\right| \tag{1}$$

We define word-level accuracy similarly. The number of words whose part-of-speech annotation was agreed upon by both humans and individual tools divided by the number of word-level human annotations. The equation follows:

$$\left|H_{word} \cap T_{word}\right| \div \left|H_{word}\right| \tag{2}$$

To calculate $H_{gp} \cap T_{gp}$, we compare grammar pattern strings for individual identifiers from the human annotations with the corresponding tool annotations for the same identifier (i.e., using string matching) and take only exact string matches. To calculate $H_{word} \cap T_{word}$, we compare grammar pattern strings for individual identifiers the same way **except** we only look for exact string matches in the individual part-of-speech tags (i.e., for corresponding words) within the grammar pattern instead of requiring the full grammar pattern to match. For example, given two identifiers: *get token string* and *set factory handle*, which have a human annotated grammar pattern of *V NM N*, if one tagger gives us the pattern *NM NM N* then we would say that there is no grammar pattern intersection; the humans and tool gave different grammar patterns. However, there is an intersection here if we only look at individual part-of-speech tags. Both the tagger and humans annotated NM and N in the last two words of each identifier. Thus, these are considered matches and would be found in $H_{word} \cap T_{word}$. If a second tagger provided the grammar pattern *V NM N*, then this would be found in $H_{gp} \cap T_{gp}$ and the individual annotation matches would be in $H_{word} \cap T_{word}$.

3.3. Data Collection

We collected our identifier set from twenty open-source systems. We chose these systems to vary in terms of size and programming language while also being mature and having their own development communities. We did this to make sure that the identifiers in these systems have been seen by multiple contributors and that the identifiers we collected are not biased toward a specific programming language. There are two reasons for choosing identifiers from multiple languages. 1) We want to know what patterns cross-cut between languages, such that most Java/C/C++ developers are familiar with and leverage these patterns. Focusing on just one language might mean the patterns we find are not common to developers outside of the chosen language. 2) Many systems are written in more than one language, and it is important to understand how well part-of-speech tagging technologies will work on these systems. Thus, running our study systems written in different programming languages helps us study part-of-speech tagger results in an environment leveraging multiple programming languages. This does not mean that none of our patterns are biased to one language or system, but that the most frequent patterns are less likely to be; we confirm cross-cutting patterns in Section 4.

We provide the list of systems and their characteristics in Table 5. The systems we picked were 615 KLOC on average with a median of 476 KLOC, a min of 30 KLOC, and a max of 1,800 KLOC. Further, most of these systems have been in active development for the past ten years or more and all of them for five years or more. The younger systems in our set are popular, modern programs.

Table 5: Systems used to create dataset

Name	**Size (kloc)**	**Age (years)**	**Language(s)**
junit4	30	19	Java
mockito	46	9+	Java
okhttp	54	6	Java
antlr4	92	27	Java/C/C++/C#
openFrameworks	130	14	C/C++
jenkins	156	8	Java
irrlicht	250	13	C/C++
kdevelop	260	19	C/C++
ogre	370	14	C/C++
quantlib	370	19	C/C++
coreNLP	582	6	Java
swift	601	5	C++/C
calligra	660	19	C/C++
gimp	777	23	C/C++
telegram	912	6	Java/C/C++
opencv	1000	19	C/C++
elasticsearch	1300	9	Java
bullet3	1300	10+	C/C++/C#
blender	1600	21	C/C++
grpc	1800	5	C++/C/C#

For example, Swift is a well-known programming language supported by Apple, Telegram is a popular messaging app, and Jenkins is a popular development automation server. Because we are trying to measure the accuracy of part-of-speech techniques and understand common grammar patterns, our goal is not necessarily to study only high-quality identifier names, but to study names that are closely representative of the average name for open-source systems. Additionally, we remove identifiers that appear in test files, in part because they sometimes have specialized naming conventions (e.g., include the word 'test', 'assert', 'should', etc). We exclude test-related identifiers by ignoring annotated test files and directories; any directory, file, class, or function containing the word *test*. While it is possible that identifiers in test code have similar grammar patterns to identifiers outside of test code, it is also possible that they do not. We did not want to risk introducing divergent grammar patterns. We think it would be appropriate to study test identifier grammar patterns separately to confirm their similarity, or dissimilarity, to other identifiers.

To collect the 1,335 identifiers, we scanned each of our 20 systems using srcML [41] and collected both identifier names/types and the category that they fell into (e.g., class, function). Then, for each category, we randomly selected one identifier from each system using a round-robin algorithm (i.e., we picked a random identifier from system 1, then randomly selected an identifier from system 2, etc. until we hit 267). This ensured that we got either 13 or 14 identifiers from each system (267/20 = 13.35) per category and mitigates the threat of differing system size.

We provide all data that was part of this study on our webpage [6] for replication, extension, and for use in further evaluation of part-of-speech taggers for source code.

[6] https://scanl.org/

Table 6: Top 5 patterns in dataset along with frequency of each pattern and % of the set represented by that pattern

Attribute Names							
Humans		Posse		Swum		Stanford	
NM N	78 (29.2%)	N N	82 (30.7%)	NM N	122 (45.7%)	N N	59 (22.1%)
NM NM N	34 (12.7%)	N N N	35 (13.1%)	NM NM N	63 (23.6%)	N N N	26 (9.7%)
NM NPL	26 (9.7%)	NM N	31 (11.6%)	NM NM NM N	32 (12%)	NM N	18 (6.7%)
N	16 (6%)	N	26 (9.7%)	N	27 (10.1%)	V N	16 (6%)
NM NM NM N	11 (4.1%)	NM N N	16 (6%)	NM NM NM NM N	11 (4.1%)	N NPL	16 (6%)
Declaration Names							
NM N	116 (43.4%)	N N	112 (41.9%)	NM N	164 (61.4%)	N N	60 (22.5%)
NM NM N	43 (16.1%)	NM N	41 (15.4%)	NM NM N	69 (25.8%)	NM N	33 (12.4%)
NM NPL	30 (11.2%)	N N N	30 (11.2%)	NM NM NM N	17 (6.4%)	V N	24 (9%)
NM NM NPL	8 (3%)	NM N N	19 (7.1%)	N	6 (2.2%)	N N N	21 (7.9%)
NM NM NM N	6 (2.2%)	N	6 (2.2%)	DT NM N	4 (1.5%)	N NPL	15 (5.6%)
Parameter Names							
NM N	122 (45.7%)	N N	96 (36%)	NM N	155 (58.1%)	N N	62 (23.2%)
NM NM N	36 (13.5%)	NM N	38 (14.2%)	NM NM N	63 (23.6%)	V N	32 (12%)
N	21 (7.9%)	N N N	28 (10.5%)	N	30 (11.2%)	NM N	31 (11.6%)
NM NPL	20 (7.5%)	N	23 (8.6%)	NM NM NM N	11 (4.1%)	N N N	20 (7.5%)
NM NM NPL	12 (4.5%)	NM N N	16 (6%)	DT N	4 (1.5%)	N	15 (5.6%)
Function Names							
V NM N	46 (17.2%)	V N N	49 (18.4%)	V NM N	66 (24.7%)	V N N	42 (15.7%)
V N	26 (9.7%)	V N	40 (15%)	V N	41 (15.4%)	V N	33 (12.4%)
V NM NM N	17 (6.4%)	V NM N	20 (7.5%)	V NM NM N	33 (12.4%)	V N N N	19 (7.1%)
NM N	15 (5.6%)	V N N N	15 (5.6%)	V NM NM NM N	14 (5.2%)	N N N	8 (3%)
V NM NPL	10 (3.7%)	V NM N N	10 (3.7%)	NM N	13 (4.9%)	NM N	8 (3%)
Class Names							
NM N	81 (30.3%)	N N	76 (28.5%)	NM NM N	98 (36.7%)	N N	71 (26.6%)
NM NM N	72 (27%)	N N N	59 (22.1%)	NM N	94 (35.2%)	N N N	69 (25.8%)
NM NM NM N	16 (6%)	NM N N	23 (8.6%)	NM NM NM N	39 (14.6%)	N N N N	23 (8.6%)
N	14 (5.2%)	NM N	19 (7.1%)	N	16 (6%)	N	13 (4.9%)
PRE NM N	10 (3.7%)	N N N N	15 (5.6%)	NM NM NM NM N	8 (3%)	NM N	9 (3.4%)

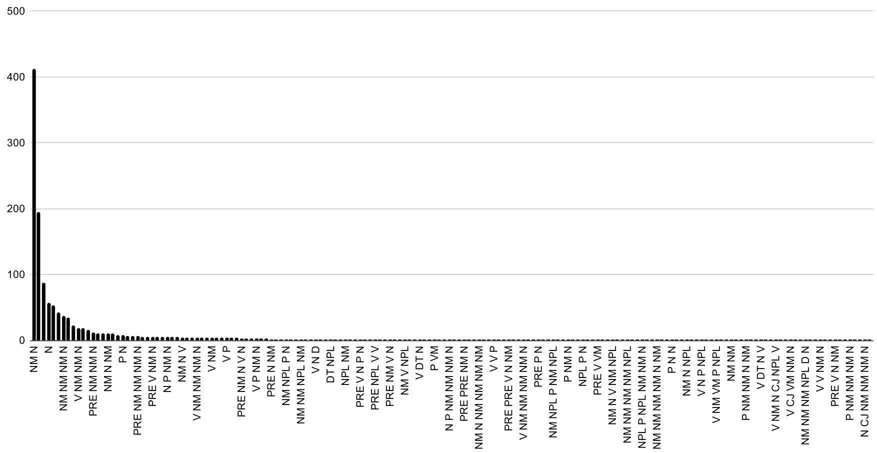

Figure 2: Distribution of unique grammar pattern frequency over entire dataset – not all unique patterns are shown due to space.

4. Evaluation

Our evaluation aims to 1) establish and explore the common and diverse grammar pattern structures found in identifiers. 2) Use these structures to investigate the accuracy, strengths, and weaknesses of approaches which generate grammar patterns with an eye toward establishing and improving their current ability. And 3) leverage the grammar patterns we discover to discuss the ways in which words, as part of a larger identifier, work together to convey information to the developer. We address these concerns in the research questions that follow. For the discussion of RQs below, the *+* symbol means "one or more" and the *** symbol means "zero or more" of the annotation to the left of the symbol; similar to how they are used in regular expressions.

4.1. RQ1: What are the most frequent human-annotated grammar patterns and what are the semantics of these patterns?

Table 6 contains data from the human-annotated set separated into categories based on the location of the identifier in code. The table shows the top

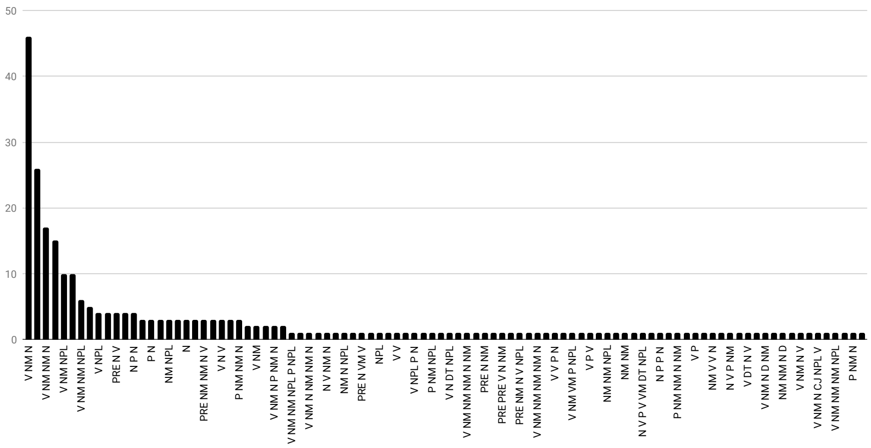

Figure 3: Distribution of unique method grammar pattern frequency – not all unique patterns are shown due to space.

five grammar patterns for each category and each of the three taggers we used to generate the grammar patterns. In addition, Figure 2 shows the distribution of unique grammar patterns across all identifiers. It shows that a minority of the total grammar patterns found in the set are repeated frequently, while the majority occur only once. Due to the fact that functions had the highest number of unique grammar patterns (Table 4), we also show the distribution of unique function grammar patterns in Figure 3. The distribution is largely similar to the prior figure with all unique grammar patterns, but the most common function grammar patterns are different than the general set due to the semantics being conveyed by function names (e.g., use of verbs to convey actions) versus other identifiers. There are three grammar patterns from which most frequent grammar patterns we will discuss are derived. These are shown and described in Table 7. Specifically, they are the verb phrase, noun phrase, and plural noun phrase patterns. We now discuss the most common grammar patterns found in our data set.

Grammar Pattern *PRE* NM+ N*: Looking at Table 6, the *NM N* instance of this pattern appears in the top five most frequent in each category,

Table 7: Grammar patterns from which other grammar patterns are frequently derived.

Grammar Pattern	Pattern Semantics
NM+ N	Noun phrase pattern: One or more noun modifiers (adjectives or noun-adjuncts) that modify a single head-noun. Noun modifiers are used to modify developers' understanding of the entity referred to by the head-noun. Because identifier names are typically associated with a single entity (e.g., a list entity, a character entity, a string entity), the head-noun typically refers to this entity. The noun modifiers, which are typically to the left the head-noun, are used to specify characteristics, identify a context, or otherwise help the developer gain a stronger, more specific understanding of what this entity embodies.
V NM+ N	Verb phrase pattern: A verb is followed by a noun phrase. Verb phrases in source code combine the action of a verb with the descriptiveness of noun phrases; the verb specifies the action and the noun phrase contains both the entity (i.e., the head-noun) which will be acted upon as well as noun modifiers which specify characteristics, identify a context, or otherwise help the developer gain a stronger, more specific understanding of what this entity embodies.
NM+ NPL	Plural noun phrase pattern: Similar to a regular noun phrase category (NM+ N), but the head-noun is plural instead of singular. This is sometimes purposeful; used to refer to arrays, lists, and other collection types or used to refer to the multiplicity of heterogeneous data groupings (i.e., classes/objects).

and it is the most ubiquitous pattern in our dataset. Noun modifiers are used to modify developers' understanding of the entity referred to by the head-noun (Table 7). **Examples**: *factory class*, *auth result*, and *previous caption.* These identifiers embody the noun phrase concept. The specific entities that they reference are *class*, *result*, and *caption* respectively. While the noun modifiers (factory, auth, and previous) specify some characteristics of the entity they comes before. The *class* is not just any kind of class; it is a *factory* class, the *result* is specifically the consequence of some form of *authentication*, the *caption* being referred to is the *previous* in relation to some other caption. Adding more noun modifiers emphasizes this effect. **Examples:** *max buffer size* uses *max* and *buffer* to describe characteristics of the *size*, which is the head-noun. The same applies to *previous initial value* and *network security policy.*

This pattern is found in all categories, but it is somewhat out of place in the Function Name category, since function names tend to contain a verb. We manually investigated these instances and found that many of them are functions with an implied verb. **Examples:** *deep Stub.* It contains no verb, but its behavior is to return (i.e., *get*) a *deep stub* object. Another example is the *lower Boundary Factor* function which returns the *lower boundary factor* based on a parameter. Many of these functions are getters or setters where the *get* or

Table 8: Frequency at which identifiers with a verb in their grammar pattern also have a boolean type and frequency at which identifiers with a boolean type have a verb in their grammar pattern

	Identifier Contains Verb	Identifier Has Boolean Type	Identifier Has Boolean Type & Contains Verb	% of Boolean Identifiers Containing a Verb	% of Verb Identifiers With a Boolean Type
Parameters	24	28	22	92%	79%
Declarations	21	21	18	86%	86%
Attributes	23	24	18	78%	75%

set is not in the name of the function.

The PRE* at the front of this pattern is optional, but appears frequently in the Class Name category. *PRE* represents preambles, as defined in Section 2. They are important to detect because if we cannot identify preambles automatically, tools may assume that the preamble somehow augments developer understanding of the head noun– which it does not. Therefore, preamble detection can help tools avoid making mistakes in interpreting words in an identifier. We discuss more about preamble patterns in Section 4.2.

The ubiquity of noun-phrase patterns suggests that part-of-speech taggers should predict *NM* on unknown, non-numeric tokens which are not the last token in the identifier– for the last token, *N* would be a better prediction.

Grammar Pattern *V NM+ N*: This pattern and its extensions are most common in functions. This is a verb phrase pattern, where a verb is followed by a noun phrase (Table 7). **Examples**: *check gl support*, *resize nearest neighbor*, and *get max shingle diff*. In *check gl Support*, the noun phrase specifies what we are interested in (openGL support), *check* tells us that we are checking a condition– specifically that *gl support* is available. The same applies for the other two identifiers; specifying which neighbor to *resize* in *resize nearest neighbor* and the nature of the *diff[erence]* to return (i.e., *get*) in *get max shingle diff*.

One characteristic we observed about this and other verb-based patterns is that, when it appears outside of function names, it tends to be for an identifier with a boolean type or a type which can be treated as boolean (e.g., integer). **Examples**: *add bias to embedding*, *is first frame*, and *will return last parameter*.

Table 9: Frequency at which identifiers ending with a plural also have a collection type and frequency at which identifiers with a collection type end with a plural

	Identifier has Collection Type	**Identifier Ends with Plural**	**Identifier Ends with Plural & Has Collection Type**	**% of Identifiers w/Collection Type & End With Plural**	**% of Identifiers w/Plural Ending & Collection Type**
Parameters	49	42	21	43%	50%
Decls	56	49	24	43%	49%
Attributes	43	61	31	72%	51%
Functions	21	44	10	48%	23%

This is because boolean variables act like predicates; asking a question whose answer is true or false (e.g., add bias to [the] embedding?, is [this] the first frame?, will [this] return [the] last parameter?). Other researchers have made this observation [22, 23], but only one reports quantity [31]. In Table 8, we show two things: 1) the percentage of all identifiers in the dataset with a at least one verb in their grammar pattern and a type which can be interpreted as boolean. 2) the percentage of all identifiers in the dataset with a boolean type that also have a at least one verb in their grammar pattern.

The observation is supported, especially amongst parameter and declaration-statement identifiers where 92% (22/24) and 86% (18/21) respectively of all identifiers with grammar pattern containing a verb also have a boolean type. Likewise, of all parameters and declaration-statement identifiers with a boolean type, 79% (22/28) and 86% (18/21) respectively contain a verb in their grammar pattern. Given that a part-of-speech tagger can be made aware of a given identifier's type, this trend should be useful in helping properly annotate boolean variables as well as suggesting higher-quality names for boolean variables.

Grammar Pattern *V* NM+ NPL*: This pattern has two configurations. It is either a plural noun phrase pattern or a plural verb phrase pattern (Table 7). Assuming that the use of a plural must be significant somehow, since some sources advise using plural identifiers for collections and certain types of classes [42], we analyzed our dataset to see if identifiers which have a plural head-noun were more likely to have a type which indicated a collection (e.g., list, array) type. To do this, we examine the type name for each identifier and record if it contains the words: *list, map, dictionary, collection, array, vector, data, or set.* We additionally record if the type name is plural (e.g., ending in -s) or if the identifier has square brackets (i.e., []) next to it. We then manually check these to see if they were really collection types. The results of this investigation are in Table 9. This table shows two perspectives on the data: 1) how many identifiers with a collection type also have a plural head-noun. 2) how many identifiers that have a plural head-noun also have a collection type.

We found that of all identifiers with a collection type, 43%, 43%, 72%, and 48% of Parameter, Declaration-statement, Attribute, and Function identifiers, respectively, are also plural. Additionally, of all identifiers that are plural, 50%, 49%, 51%, and 23% of Parameter, Declaration-statement, Attribute, and Function identifiers, respectively, have a collection type. Similar to booleans, we find that there is a trend– particularly for attribute identifiers with a collection type, which had the highest likelihood of also being plural. While this is not always the majority case, it does suggest an interesting direction for future research into the use of plural names to convey the use of collections in different types of identifiers. **Examples**: *mkt factors*, *num cols*, and *child categories.* The *mkt factors* identifier is a plural noun phrase that represents a collection entity (e.g., a list or array) of market factors. The *num cols* identifier represents the number of columns for some entity (e.g., a matrix), and the *child categories* identifier represents a set of categories with a parent-child relationship to a super-category.

The weakest correlation between use of plural noun phrases and collection type is found in the Function Name category as part of a plural verb phrase. Instead of representing a collection, Function identifiers following a plural verb phrase pattern often allude to the multiplicity of the data being operated on, and not necessarily returned. A few examples from our data set are: *Object getRawArguments(...)*, *void validateClassRules(...)*, and *Object getActualValues(...).* In all three cases, while a collection is not explicitly returned, the functions refer to an entity which represents multiple heterogeneous data (i.e., an object) that is then returned, part of the calling object, or incoming data as a parameter. This behavior is not unique to functions, but more frequent in function identifiers versus other identifiers. Therefore, we should be cautious when making assumptions.

Grammar Pattern *V N*: This pattern represents an action applied to or with the support of an entity represented by the head-noun. This pattern commonly represents function names or boolean identifiers, similar to the *V NM N* pattern from which it differs only due to the lack of a noun-adjunct. **Example:** *read subframe* has the grammar pattern *V N* and names a function which reads a subframe object.

Grammar Pattern *N*: A single noun, trivially a head-noun, represents a singular entity and could be considered a basic case of the noun phrase pattern (i.e., with no NMs). **Example**: *client* and *usage*. Poorly split, or purposefully not split, abbreviations are automatically tagged as a single noun. This phenomenon is not necessarily incorrect as it could be considered a collapsed noun-phrase pattern (e.g., *max buffers* abbreviated as *mb* which then gets annotated as a noun). Developers familiar with the abbreviations may even prefer this form and certain, extremely well known, abbreviations may even be treated as singular nouns during comprehension. For example, IPV4, MP3, HTTP, HTML are common abbreviations which are more common to see/read than their expansions (e.g., MP3 is rarely expanded).

Summary for RQ1: Table 6 contains the most frequent grammar patterns in the human-annotated set. We identified five patterns by looking at how frequently they occurred in our human-annotated dataset. By far, the most ubiquitous pattern we found was the noun phrase (*NM+ N*) pattern as it appears in every category. We also found that verb phrase (*V NM+ N)* patterns are most common in function names but also appear in other types of identifiers; specifically those with a boolean type, and that plural noun phrases (*NM+ NPL*) have a somewhat heightened chance of representing collection identifiers. Results indicate that 1) due to the ubiquity of noun phrase patterns, part-of-speech taggers should predict *NM* on unknown, non-numeric tokens that are not the right-most token in the identifier; *N* is a better prediction for the right-most token, as it is likely the head-noun. 2) our data supports the observed,

Table 10: Frequency of per-tag agreement between human annotations and tool annotations

Part of Speech	**Human Annotations**	**Posse**	**% Agreement w/ Humans**	**Swum**	**% Agreement w/ Humans**	**Stanford**	**% Agreement w/ Humans**
NM	1604	373	23.25%	1508	**94.01%**	252	15.71%
N	1141	1025	89.83%	976	85.54%	1064	**93.25%**
V	305	205	67.21%	171	56.07%	233	**76.39%**
NPL	238	0	0.00%	0	0.00%	171	**71.85%**
PRE	105	0	0.00%	2	**1.90%**	0	0.00%
P	94	59	62.77%	28	29.79%	85	**90.43%**
D	27	0	0.00%	5	18.52%	27	**100.00%**
DT	15	6	40.00%	13	**86.67%**	9	60.00%
VM	13	0	0.00%	0	0.00%	9	**69.23%**
CJ	8	0	0.00%	0	0.00%	4	**50.00%**
Total words:	***3550***	***1668***		***2703***		***1854***	

heightened appearance of verbs in boolean variables from prior work. 3) while there is a link between identifier names containing a plural and collection data types, the plural is sometimes used to reference multiplicity of related, heterogeneous data (i.e., class member data), particularly for function identifiers. This presents an opportunity to support developers in how, and when, to use plurals. The grammar patterns we generated highlight how the name of an identifier is influenced by the semantics of the language and gives us a glimpse into how developers use words in an identifier to comprehend their code.

4.2. RQ2: How accurately do the chosen taggers annotate grammar patterns and individual tags?

We compare the output of the three part-of-speech taggers in this study with the manually-annotated grammar patterns in order to calculate the accuracy of each tagger at both the level of grammar patterns and the level of individual part-of-speech tags. Please refer to Section 3.2 for an explanation of how we calculate accuracy. Starting with Table 10, which contains our per-tag (i.e., word-level) accuracy analysis, we observe that Swum had the highest agree-

Table 11: Percentage of tool-annotated grammar patterns which fully match human-annotated grammar patterns

Category	Posse	Swum	Stanford (V)	Stanford (NM)	Stanford+I (V)	Stanford+I (NM)
Parameters	58 (21.7%)	181 (67.8%)	63 (23.6%)	71 (26.6%)	N/A	N/A
Declarations	47 (17.6%)	163 (61%)	51 (19.1%)	54 (20.2%)	N/A	N/A
Attributes	45 (16.9%)	135 (50.6%)	45 (16.9%)	51 (19.1%)	N/A	N/A
Functions	66 (24.7%)	134 (50.2%)	60 (22.5%)	58 (21.7%)	68 (25.5%)	67 (25.1%)
Classes	35 (13.1%)	180 (67.4%)	26 (9.7%)	31 (11.6%)	N/A	N/A

ment with the human annotations with respect to noun modifiers, determiners, and preambles. Stanford had the highest agreement with respect to everything else. Posse never outperformed both the other two in a single category, but did perform better than Stanford at annotating noun modifiers and better than Swum at annotating nouns, prepositions, and verbs. The numbers here indicate that Stanford has the best all-around performance when we are looking at accuracy on individual part-of-speech annotations, as it is able to detect a broader range of them more accurately than either Swum or Posse. However, while Stanford had the highest accuracy on the largest number of part-of-speech tag types, Swum tagged the highest number of raw words correctly with 2,703 correct annotations versus Stanford's 1,854. The results indicate areas of strength for each tagger; their combined output may increase their overall accuracy.

With this context in mind, we will now look at the agreement between the tagger-annotated and human-annotated grammar patterns (i.e., identifier-level accuracy analysis). This is shown in Table 11. We broke Stanford down into several columns in this table to see how its accuracy changes when we configure it differently. The configurations are as follows: We add an *I* before function names before applying Stanford tagger. Additionally, some types of verbs can be considered adjectives in different contexts. Thus, we test the accuracy of Stanford under either assumption; the verb being used as a verb or being used as an adjective. Details of both all configurations for Stanford are given in Section 2.

This data shows that Swum had the highest agreement with the human-provided annotations at the level of grammar patterns. Stanford had the second-highest agreement on average when we assume its best configuration– though, Posse has a higher agreement with the human annotations in the Classes category regardless of Stanford's configuration. Swum's accuracy ranged between 50.2% and 67.4% while Posse and Stanford's ranged between 13.1% - 24.7% and 9.7% - 26.6% respectively. The difference in the results between Tables 10 and 11 are interesting in that Stanford has the best performance in Table 10 but under-performs Swum by a large margin in Table 11. The reason for this difference is the ubiquity of noun modifiers in identifier names. Even though Stanford is more accurate on a larger set of part-of-speech tag categories, it under-performs on noun modifiers compared to Swum (15.71% accuracy for Stanford vs 94.01% for Swum), which consistently annotates noun modifiers correctly. Noun modifier is the most frequent annotation (with a frequency of 1,604 per Table 10); Swum gets 1508 of these correct while Stanford gets 252, meaning Stanford missed 1256 words that Swum got correct. If we combine this with the fact that Swum's performance on the second-most-common annotation, nouns, is much closer to Stanford's (85.54% accuracy for Swum vs 93.25% for Stanford) than Stanford's performance is to Swum's on noun modifiers, it makes sense that, when looking at accuracy on annotating full identifier name grammar patterns (i.e., Table 11), Swum outperforms Stanford despite Stanford's high annotation accuracy on most other part-of-speech tag types. In short, Stanford is more likely to get the very common *NM+ N* pattern incorrect due to mis-annotating NM compared to Swum, which will occasionally mis-annotate N, but not as frequently as Stanford will mis-annotate NM.

Using this data, we can also confirm that Stanford's accuracy is increased in method names when appending and *I* to the beginning of the name. This causes it to more accurately identify verbs. Additionally, Stanford's accuracy increases in methods when the verb specializations discussed in Section 2 are assumed to be verbs and increases in every other category when they are assumed to be noun modifiers.

To understand more about the differences in agreement between the humans and taggers, we identified which patterns were most frequently incorrectly generated for each tagger, in part to understand each tagger's weaknesses. These patterns represent paths toward significantly improving the accuracy of part-of-speech taggers for source code identifiers. Table 12 gives the top five most frequently mis-annotated grammar patterns per category for each tagger used in our study.

To contextualize the details we discuss below, we quickly summarize some of the core problems found in the part-of-speech tagger output using the data in Table 12. Posse has trouble generating noun phrase and verb phrase patterns (i.e., *NM+ N* and *V NM+ N* respectively), including plural noun phrases such as *NM+ NPL*. The reason for this is that Posse does not generally identify noun modifiers in sequences greater than 1 (i.e., it may annotate *NM N*, but never *NM NM N*). Even on sequences with only one noun modifier, it tends to prefer annotating noun modifiers as a noun. This problem is more pronounced in Stanford, which generally missed more noun phrase and verb phrase patterns than Posse. This makes sense as Stanford did not annotate noun modifiers very well (Table 10). However, Stanford is very good at identifying plurals; Swum and Posse never identified plural words correctly in the dataset (Table 10). This is why plural noun and verb phrase patterns are both frequently mis-annotated by both Swum and Posse. We will now focus our discussion around specific, commonly mis-annotated grammar patterns and discuss tagger annotations in the context of each.

Grammar Pattern *NM+ N*: Swum was the best tagger at identifying noun phrase patterns because it very accurately recognized noun modifiers. It did overestimate noun phrase patterns due to over-annotating noun modifiers where they do not belong. **Example**: *rotation Per Second* has a pattern *N P N*. Swum mis-annotates two out of three words by annotating this identifier as *NM NM N*; failing to recognize the fact that *Per* is a preposition in this context.

Posse and Stanford have a harder time with noun phrase patterns and tend to use a noun instead of a noun modifier. **Example**: *cache entity* and *root ptr* are

Table 12: Top 5 Most frequently mis-annotated grammar patterns

Attribute Names					
Posse		Swum		Stanford	
NM N	58 (26.1%)	NM NPL	26 (19.7%)	NM N	61 (28.4%)
NM NM N	31 (14.%)	NM NM NPL	9 (6.8%)	NM NM N	33 (15.3%)
NM NPL	26 (11.7%)	NPL	9 (6.8%)	NM NPL	19 (8.8%)
NM NM NM N	11 (5%)	PRE NM N	8 (6.1%)	NM NM NM N	11 (5.1%)
NM NM NPL	9 (4.1%)	PRE N	7 (5.3%)	NM NM NPL	9 (4.2%)
Declaration Names					
NM N	84 (38.2%)	NM NPL	30 (28.8%)	NM N	88 (41.3%)
NM NM N	39 (17.7%)	NM NM NPL	8 (7.7%)	NM NM N	42 (19.7%)
NM NPL	30 (13.6%)	N D	5 (4.8%)	NM NPL	26 (12.2%)
NM NM NPL	8 (3.6%)	V N	5 (4.8%)	NM NM NPL	8 (3.8%)
NM NM NM N	6 (2.7%)	N P N	4 (3.8%)	NM NM NM N	6 (2.8%)
Parameter Names					
NM N	92 (44%)	NM NPL	20 (23.3%)	NM N	92 (46.9%)
NM NM N	35 (16.7%)	NM NM NPL	12 (14.%)	NM NM N	35 (17.9%)
NM NPL	20 (9.6%)	V N	6 (7%)	NM NPL	15 (7.7%)
NM NM NPL	12 (5.7%)	NPL	5 (5.8%)	NM NM NPL	12 (6.1%)
NPL	5 (2.4%)	NM N	3 (3.5%)	N	6 (3.1%)
Function Names					
V NM N	33 (16.4%)	V NM NPL	10 (7.5%)	V NM N	42 (21.1%)
V NM NM N	15 (7.5%)	NM N	6 (4.5%)	V NM NM N	17 (8.5%)
V NM NPL	10 (5%)	V NM NM NPL	6 (4.5%)	NM NM N	10 (5%)
NM NM N	10 (5%)	N V	5 (3.8%)	NM N	8 (4%)
NM N	9 (4.5%)	V NPL	4 (3%)	V NM NPL	8 (4%)
Class Names					
NM NM N	70 (30.2%)	PRE NM N	10 (11.5%)	NM N	72 (30.5%)
NM N	63 (27.2%)	NM NPL	8 (9.2%)	NM NM N	69 (29.2%)
NM NM NM N	16 (6.9%)	NM N NM	7 (8%)	NM NM NM N	16 (6.8%)
PRE NM N	10 (4.3%)	NM NM N	5 (5.7%)	PRE NM N	10 (4.2%)
NM NPL	8 (3.4%)	PRE NM NM N	5 (5.7%)	NM N NM	7 (3%)

both given an *N N* pattern by Stanford and Posse. While annotating using noun is not wholly inappropriate, these nouns play a double role of both identifying an external concept which exists as its own object (the root, the cache; both of which are nouns) and using this concept to modify the head-noun-of-interest (e.g., the entity, the pointer) so the developer fully understands what they are dealing with; root and cache are nouns being used as noun modifiers and not pure nouns. An easy way to see this adjectival relationship is to add a dash between the words; root-pointer, cache-entity. Root describes the pointer, cache describes the entity.

Grammar Pattern *NM* NPL*: Posse and Swum do not detect plurals, while Stanford is very good at detecting them (see Table 10). Stanford tends to get noun plurals individually correct even when it would mistakenly annotate a noun modifier as a noun, which is why *NM NPL* was still one of the most common patterns for Stanford to mis-annotate. **Example**: *num active contexts* has a pattern *NM NM NPL* due to the plural at the end. Swum does not recognize the plural and gives it a *NM NM N* pattern. Posse gave it an *N NM N* pattern and Stanford gave it a *N NM NPL* pattern. Thus, Stanford and Swum were both nearest to the correct solution despite both being incorrect. Stanford did not have trouble with *NPL* patterns when there were no noun modifiers. This suggests a very good way that tagger annotations may be combined; Stanford can identify noun plurals accurately and Swum can identify noun-modifiers accurately. In general, this was the hardest, frequently observed pattern for any individual tagger to annotate completely correctly.

Grammar Pattern *V NM+ N*: The deciding factor in mis-annotating verb phrase patterns tended to be noun modifiers and plural nouns. Generally speaking, the taggers agreed with the human annotators on the position of the verb in method names between 56% and 76% of the time. This contrasts with how often they agreed on the full human annotation in function names (Table 11); 24.7% of the time for Posse, 50.2% of the time for Swum, 25.5% of the time for Stanford. All taggers still have problems determining the correct verb, but detecting noun modifiers is the bigger issue.

Posse and Stanford had the most trouble with this pattern; it is not in Swum's top 5. **Examples**: *reset meta class cache* and *set project naming strategy*; both with a human-annotated grammar pattern of *V NM NM N*. Swum agreed with the human annotators on both; Stanford annotated both with *V N N N*; and Posse annotated these as *V N N N* and *V N NM N* respectively.

Grammar Pattern *V N*: This pattern tends to be mis-annotated when the part-of-speech taggers could not determine which (if any) word in the identifier was a verb. One of the most common situations for this was identifiers with a boolean type. Posse and Swum tend to expect that there is a verb when they know that they are looking at a function name, but in non-function identifiers they are less likely annotate using verb. For example, the *write root* identifier has a human-annotated pattern of *V N*. Stanford agrees with the human-annotated pattern, but Swum mis-annotates it as *NM N* and Posse as *N N*.

Grammar Pattern *V NM* NPL*: This pattern is similar to *V N* in that one of the biggest problems the taggers had was annotating the verb. However, this pattern was more trouble for Posse and Swum than Stanford due to the inclusion of a plural– Stanford detects plurals well, but neither Swum or Posse are able to determine when a word is in a plural form.

Grammar Pattern *PRE...*: We will discuss all patterns with a preamble here (i.e., the ... could be any other pattern, such as noun phrase or verb phrase). Preambles were difficult for all taggers to deal with. Swum rarely detects preambles while Posse and Stanford do not detect them at all. Generally, a preamble is mis-annotated as noun for Posse and Stanford or noun modifier for Swum. The problem with detecting preambles is that some prior information is required– a tagger needs to scan the code and/or be able to recognize naming conventions such as Hungarian [38], to determine which character sequences are being used as preambles. There are also cases where it is not clear that a frequent character sequence is a preamble. **Examples**: the GRPC project tends to append *grpc* before many of its identifiers– *grpc json writer value string* has a grammar pattern of *PRE NM NM NM N*. A scan of GRPCs code could identify this as a preamble. *XML* is sometimes used frequently at the beginning of function names such as *xmlWriter, xmlReader*. It may look like a preamble in some systems due to how frequently it appears at the beginning of an identifier, but is not a preamble (Section 2) because it specializes our understanding of the words *reader* and *writer*. This suggests that some domain knowledge is additionally required to correctly determine when a character sequence is a preamble.

Grammar Pattern *N P N*: This is a prepositional phrase grammar pattern. Swum and Posse had difficulty identifying prepositions while Stanford was effective at it. Stanford tends to do well on this pattern; it usually annotated the preposition correctly but would occasionally mistake noun for verb. This pattern is one of the less common ones in our dataset, but is an example of a pattern on which Stanford is more accurate than Swum or Posse.

Grammar Pattern *N D*: This pattern is a noun followed by a number. Swum and Posse cannot detect digits, thus Stanford is the only tagger that correctly identifies this pattern. Stanford had high accuracy on digits; agreeing with all digits identified by human annotators. It occasionally is unable to annotate the noun correctly; many of these cases are when there is an abbreviation or a word which is colloquial (or domain-specific) to programmers.

Summary for RQ2: The highest amount of agreement was between Swum and the human annotators; Swum's accuracy ranged between 50.2% and 67.8% while Posse and Stanford's ranged between 13.1% - 24.7% and 9.7% - 26.6% respectively. The most frequently incorrectly annotated patterns were: 1) singular noun phrases for Stanford, plural noun phrases for Swum and both for Posse. 2) plural verb phrases for Swum, singular verb phrases for Stanford, and both for Posse. 3) grammar patterns which include a preamble for all three taggers. Our results 1) indicate that it is possible to correctly annotate abbreviations without expanding them in some cases, particularly in noun phrases; this is supported by Swum's accuracy. 2) indicate that these taggers have complementary strengths and weaknesses, meaning that their output can be combined into a more accurate result than they are able to obtain individually. As a simple example, Stanford annotates noun plurals fairly accurately; this could be used to improve Swum or Posse's output on *NM NPL*, while Swum/Posse can improve Stanford's *NM* detection; causing all three taggers to get *NM NPL* correct more frequently. 3) confirm that Stanford's accuracy on functions is improved by adding *I* and that certain verb forms are more likely to be verbs when found in function names but adjectives when found in other types of identifiers. This also shows that Swum and Posse need to detect specialized verb forms in order to correctly identify when these verbs are used as verbs or adjectives. 4) show that code context and domain-specific information are very important for annotating words correctly; preambles are one of the categories that would most benefit from taggers which are able to leverage surrounding code structure, naming conventions, and domain information.

4.3. RQ3: Are there other grammar patterns that are dissimilar from the most frequent in our data, but still present in multiple systems?

We observed a number of grammar patterns that were not frequent enough to be in the top 5, but nevertheless appear in multiple systems. The question is: What are these patterns? What types of identifiers do they represent? To answer these questions, we manually looked through the set of human-annotated grammar patterns and picked patterns which occurred two or more times in any single category (i.e., function, class, etc.) and are not similar to the patterns we discussed in RQ1. This resulted in the following grammar patterns:

Grammar Pattern *NM* N P N*: This pattern is a noun phrase and a prepositional phrase combined. **Examples**: *depth stencil as texture* and *scroll id for node.* Identifiers with this pattern describe the relationship between a noun phrase on the left of the preposition and a noun phrase on the right. The noun phrase on the right and left both contain a head-noun. In this case, *stencil* and *id* are the left head-nouns (lhn) while *texture* and *node* are the right head-nouns (rhn). The preposition tells us how these head-nouns are related to one another and how this relationship defines the identifier. In *depth stencil as texture* we are told that this identifier represents the texture version of a stencil– or a conversion from stencil to texture. In *scroll id for node* we are told that this identifier represents an id for a specific node.

An example which does not include a noun modifier is *angle in radian* with a grammar pattern of *N P N*. The same concepts above apply– *angle* is the lhn and *radian* is the rhn. The preposition *in* helps us understand how their relationship defines this identifier. In this case, the identifier represents an angle using radians as the unit of measurement.

Grammar Pattern *P N*: This pattern is a preposition followed by a noun. An example of this pattern is the identifier *on connect*, which specifies an event to be fired when a connection is made. Other instances of this pattern include *with charset* and *from server*. The former is a function which takes a charset as its parameter, and the latter is a boolean which tells the developer whether certain data was obtained from a server. Identifiers in event-driven systems likely use this pattern, or its derivatives, often (e.g., onButtonPress, onEnter). This suggests that more unique grammar patterns may be obtained by studying identifiers found in systems using certain architectural, or programming, patterns.

Grammar Pattern *DT NM* NPL*: this pattern is a determiner followed by a plural noun phrase. **Example**: *all invocation matchers*. Determiners like *all* are called quantifiers. They indicate how much or how little of the head-noun is being represented. So in this case, the identifier represents a list of every invocation matcher. This pattern is familiar in that it contains a plural noun phrase, but the inclusion of the determiner quantifies the plural noun phrase more formally than if it did not include it; *invocation matchers* without *all* indicates a list of invocation matchers, but *all invocation matchers* tells us the specific population matchers included in the list. The word *all* was the most common determiner used for the identifiers that fit this pattern in our dataset. This pattern may show up more in code that deals with querying– databases or other query-able structures, where words like *all* and *any* might be useful. Further study is required to determine common contexts for determiners in source code.

Grammar Pattern *V+*: Like the other patterns derived from verbs above, this one is typically used with Boolean variables and functions. One interesting thing about this pattern is that there there is no noun for the verb to act on. When this happens in a function name, it is because the noun is contained within the function's arguments or is the calling object itself (i.e., this). **Examples**: *delete*, *do forward*, *parsing*, and *sort*. A delete function could either be applied to an argument to *delete* that argument, or to the calling object to delete some internal memory. The *do forward* function in our dataset redirects a user (i.e., *forward* is being used as synonym for the verb *redirect*), and the system it is from uses *do* to prefix methods which perform, for example, HTTP actions. The V+ pattern can also represent Boolean variables that are not function names. The *parsing* identifier is a Boolean variable in our dataset which is annotated as verb since it is asking a true or false in reference to an action– specifically, a parsing action. *Sort* represents a function where the user can supply a predicate to influence how elements are sorted by the function. This pattern may appear more often in generic libraries, where the head-noun is not supplied by the library creators due to the generic nature of the solution. Instead, the user will supply a head-noun when they use the library.

Grammar Pattern *V P NM N:* This pattern is a verb followed by a prepositional phrase. This was found only among function identifier names. **Examples**: *convert to php namespace* and *register with volatility spread*. This pattern uses a verb to specify an action on an unknown entity (e.g., the identifiers above do not specify what to *convert* or what to *register*) and uses the noun phrase on the right side of the preposition as a reference point; the specific thing to which this unknown entity will be related. The nature of this relationship is specified by the verb and preposition (i.e., convert to, register with).

While there are other grammar patterns which we do not discuss, many of them occur only once or are very similar in structure to grammar patterns that we have already discussed above.

Summary for RQ3: There are many ways to describe interesting types of program behavior and semantics. In RQ3 we have discussed less frequent, yet legitimate, grammar patterns. This adds diversity to the group of gram-

Table 13: Grammar patterns broken down by language

C Language Patterns		C++ Patterns		Java Patterns	
NM N	76 (33%)	NM N	175 (30.5%)	NM N	154 (30.1%)
NM NM N	17 (7.4%)	NM NM N	93 (16.2%)	NM NM N	81 (15.8%)
NM NPL	14 (6.1%)	NM NPL	31 (5.4%)	NM NPL	42 (8.2%)
N	11 (4.8%)	N	28 (4.9%)	V NM N	21 (4.1%)
V N	9 (3.9%)	V N	27 (4.7%)	NM NM NPL	20 (3.9%)
V NM N	7 (3%)	V NM N	25 (4.4%)	NM NM NM N	18 (3.5%)
NM NM NPL	6 (2.6%)	NM NM NM N	14 (2.4%)	N	16 (3.1%)
NM NM NM N	5 (2.2%)	PRE NM N	12 (2.1%)	V NM NPL	10 (2%)
NM V NM N	4 (1.7%)	V NM NM N	9 (1.6%)	V NM NM N	9 (1.8%)
PRE NM N V N	3 (1.3%)	NPL	9 (1.6%)	PRE NM N	7 (1.4%)

mar patterns discussed previously, and highlights the need for further research to uncover new grammar patterns. This will help ensure that we obtain an understanding of both the breadth and depth of grammar patterns used to describe different forms of program semantics and behavior. Grammar patterns which include prepositional phrases and determiners are less frequent than other patterns in our dataset. Yet, as we have pointed out, some of these patterns are copacetic with certain domains. Prepositional phrase grammar patterns are used in event-driven programming very frequently, for example. The patterns presented in RQ3 represent future directions for research; there are not enough examples of the patterns we presented in this research question to draw strong conclusions, but their existence is evidence that these grammar patterns may be common in other contexts (e.g., different architecture and design patterns). We argue that studying grammar patterns in these other contexts will result in more semantics belying those patterns than what we have discussed, or perhaps even new patterns. These domain-specific patterns would be very useful for suggesting and appraising identifier names within those contexts.

4.4. RQ4: Do grammar patterns or tagger accuracy differ across programming languages?

Programming languages have their own individual characteristics. To give a few (i.e., non-exhaustive) examples: C is a procedural language that does not support objected oriented programming, Java is an object oriented language, and C++ supports facets of object orientated programming and generic programming. Given this, we grouped grammar patterns in our data set by programming language with a goal varying the language to see if certain grammar patterns were more common to a specific language. This data is shown in Table 13, where we show the top 10 patterns C, C++, and Java. We note that most of the identifiers in our set were either C++ (573) or Java (511) identifiers; a smaller number were C (229) and C-sharp (22). We leave C-sharp out of our analysis due to the very small number of them, but still make this data available in our open dataset [7]. We include C in our analysis, but note that the results for C may not generalize as well as for C++ and Java.

The results in this table show that the identifiers found in individual languages are largely similar to one another. Most patterns that occurred more than once in any language also occurred in the other languages. To get a better look at language-specific patterns, we looked for any pattern which occurred multiple times but only in a specific language. After finding these patterns, we manually examined and picked patterns which were not reflective of system-specific naming conventions (i.e., only occurs in a single system). In general, the language-specific patterns tended to include determiners (DT), prepositions (P), or digits (D). For example, identifiers with grammar patterns including these annotations are: *all action roots (DT NM* N, Java)*, *group by context (N P N, Java)*, and *event 0 (N D, C++)*. We discussed the former two patterns in RQ3. The last pattern, *N D* is used typically as a way to distinguish two identifiers which otherwise have the same name (i.e., event0, event1, etc). We did not find any patterns which were significantly programming language-specific.

[7] https://scanl.org/

Table 14: Distribution of abbreviations and dictionary terms between different languages in the data set

Word Type	C	C++	Java	C#	Total in Dataset
Abbreviations	114 (22.6%)	223 (17.4%)	173 (14.5%)	9 (17.3%)	519
Dictionary Terms	505	1282	1192	52	3031

Table 15: Accuracy of taggers on abbreviated and non-abbreviated terms

Word Type	Posse	Swum	Stanford	Total in Dataset
Abbreviations	183 (35.3%)	345 (66.5%)	230 (44.3%)	519
Dictionary Terms	1484 (49%)	2321 (76.6%)	1624 (53.6%)	3031

However, this result is not a definitive answer on the differences (or lack thereof) in grammar patterns between programming languages. While it does show us that there is a lot of similarity in grammar patterns between languages, another way to interpret this data is that these differences are unlikely to be found without controlling for other factors. For example, the programming paradigms, architectural/design patterns, and problem domain of the systems in the dataset. If controlled for, these factors could reveal differences in the grammar patterns between different programming languages.

We then looked at the distribution of abbreviations and dictionary words between Java and C/C++ to provide more insight about the differences in identifier structure between languages. Since abbreviations may make it difficult to obtain the meaning of a word, part-of-speech taggers might annotate these words less accurately. Table 14 shows the distribution. We include C# in this table for completeness despite its low number of identifiers. To determine if a token is a full word or an abbreviation, we used Wordnet [43]. If Wordnet recognized the word, we considered it a full word. Otherwise, it is an abbrevi-

Table 16: Accuracy of part-of-speech taggers split by programming language

	Posse	**Swum**	**Stanford**
C	37 (16.2%)	116 (50.7%)	45 (19.7%)
C++	107 (18.7%)	357 (62.3%)	116 (20.2%)
Java	100 (19.6%)	305 (59.7%)	108 (21.1%)

ation. The results indicate that C and C++ tend to contain more abbreviated terms.

Given this, we then looked at the accuracy of each part-of-speech tagger on identifiers from different languages. This data is found in Table 15. All taggers had decreased performance on abbreviated terms, but Posse was the least accurate and saw the most significant decrease in performance (-14% from its full word accuracy). Finally, given this data about tagger performance on abbreviations and the distribution of abbreviations in different languages, we looked at tagger accuracy per programming language. This data is in Table 16. While Posse/Stanford performed better on Java than C/C++ systems, their performance degraded a total of 3.4% and 1.4% respectively. Swum, however, was nearly 12% less accurate on C identifiers compared to C++ identifiers. Swum also performed better on C++ identifiers versus Java, unlike the other two taggers.

Summary for RQ4: At the level of grammar patterns, while only controlling for identifier category (e.g., function name) and programming language, there does not appear to be a significant difference in the grammar patterns for Java and C/C++ identifiers. It may be the case that significant grammar pattern differences appear when controlling for more confounding factors and we believe this would be a strong direction for future work. We do note a difference in the use of abbreviations between C/C++ and Java identifiers, where Java identifiers tend to have fewer abbreviations than the former two languages. Abbreviations can hinder the accuracy of part-of-speech taggers, which we confirmed by examining the annotations given to abbreviations by the three taggers in this study; all taggers performed worse on abbreviations than on full words. However, difficulty with abbreviations did not significantly reduce Posse/Stanford's performance between programming languages, indicating that abbreviations are not the biggest problem these taggers face, while expanding abbreviations may significantly (up to 10%) improve Swum's performance.

5. THREATS TO VALIDITY

This study was done on a set of 1,335 identifiers; the largest set of open-source, manually tagged, cross-language identifiers at the time of writing. Even so, there is a threat that the annotated set contains imperfections. To mitigate this, we used cross-validation, where all annotators performed a validation step on all grammar patterns; each grammar pattern was validated by two annotators beside the original annotator. Additionally, the dataset is publicly available; future corrections are possible. We calculated that a statistically representative sample for the size of our dataset of 20 systems is 267 given a 95% confidence level and a confidence interval of 6%. We picked 95% and 6% as a trade-off between representativeness of the sample, the amount of manual labor required of the annotators, and the sample size used in prior studies.

In this study, we intended to focus only on production code. Thus, we have excluded test files. Yet, if developers violate JUnit test conventions by including non-annotated test functions in their production code, we may have picked these identifiers to be included in the dataset. This threat is mitigated by the fact that we manually examined every identifier.

We did not expand abbreviations and did not remove identifiers that contained abbreviations or non-dictionary terms. This resulted in an increased number of mistakes made by the taggers, as shown in RQ4. Abbreviations were included so we could measure the accuracy of part-of-speech taggers in a real-world environment, where abbreviation expansion might not be feasible. To help ensure that the humans did not mistake the annotation for abbreviated terms, the human annotators were allowed to look at the code. Additionally, we rely on identifier splitting techniques. While automated splitting is highly accurate, it is not perfect; we mitigated this problem by manually examining every identifier and correcting the split where there were mistakes.

We sampled identifiers from multiple programming languages to avoid biasing the names in our identifier set toward a particular programming language. While this means that the set we obtain is more likely to be generalizable, it also means that particular language-specific patterns may not be detected. To mitigate this, we present frequent patterns per language (Java, C, and C++). However, we had fewer identifiers in C than in Java and C++, meaning that the results for C may not be as generalizable as for C++ and Java. In addition, it is not always possible to identify an identifier as being a C or C++ identifier. In this study, we assume .c and .h files are for C systems while .cpp and .hpp files are for C++ systems. However, this only a naming convention; C++ programs may use .c and .h, for example. Next, while C, C++ and Java are different programming languages, they are all in similar programming paradigms (e.g., imperative languages). Our results may not extend to languages in other paradigms, such as functional languages. Finally, our analysis of the differences between grammar patterns in different programming languages does not take into account several potential confounding factors including system domain and design patterns. Thus, we acknowledge this problem and recommend directions for future research, based on our data, to mitigate this threat.

6. RELATED WORK

6.1. Rename Analysis

Arnoudova et al. [18] present an approach to analyze and classify identifier renamings. The authors show the impact of proper naming on minimizing software development effort and find that 68% of developers think recommending identifier names would be useful. They also defined a catalog of linguistic antipatterns [17]. Liu et al.[44] proposed an approach that recommends a batch of rename operations to code elements closely related to the rename. They also studied the relationship between argument and parameter names to detect naming anomalies and suggest renames [45]. Peruma et al. [21] studied how terms in an identifier change and contextualized these changes by analyzing commit messages using a topic modeler. They later extend this work to include refactorings [19] and data type changes [46] that co-occur with renames.

These techniques are concerned with examining the structure and semantics of names as they evolve through renames. By contrast, we present the structure and semantics of names as they stand at a single point in the version history of a set of systems. Rename analysis and our work are complimentary; our analysis of naming structure can be used to help improve how these techniques analyze changes between two versions of a name, in part by helping to improve off-the-shelf part-of-speech taggers. For example, Arnaoudova's [18] work depends on Wordnet and Stanford; we have shown that Stanford needs significant help to accurately annotate identifiers. Beyond part-of-speech tagging, our work also highlights many different grammar patterns. These patterns can be used to improve our understanding of how names evolve over time by examining how the patterns evolve.

6.2. Identifier Type and Name Generation

There are several recent approaches to appraising identifier names for variables, functions, and classes. Kashiwabara et al. [13] use association rule mining to identify verbs that might be good candidates for use in method names. Abebe [16] uses an ontology that models the word relationships within a piece of software. Allamanis et al. [14] introduce a novel language model called the Subtoken Context Model. One thing these approaches have in common is the use of frequent tokens and source code context to try and generate high-quality identifier names. In contrast, in this paper, we manually analyze identifier names to understand their structure and how that structure affects the role of individual words within the structure. A better understanding of identifier structure in different contexts can synergize with automated techniques like those above by helping them generate structurally consistent identifiers. One thing that automated techniques struggle with is generating identifiers with words which are not present in the surrounding code (i.e., neologisms [14]); a stronger understanding of naming structure combined with better part-of-speech tagging and word relationships in software (e.g., [47]) will allow these techniques to generate higher-quality names by providing guidance on what grammatical structure a new identifier should use so that new words can be picked to satisfy that structure. A stronger understanding of grammatical structure in identifiers can also help researchers train their approaches more effectively.

There has also been work in reverse engineer data types from identifiers [48, 49], focusing on using identifier names in languages such as Javascript to determine the type of the identifier. This relates most closely to our observations about List and Boolean identifiers. For example, Malik et al [48] observe that some identifiers are more likely to give away their type and their example is boolean functions, which commonly start with the words "is" or "has" and thus can be inferred to be boolean. The primary difference in our work is that the type is already available in our chosen languages (whereas they focus on Javascript), so we do not need to infer it. Instead, we are more interested in how certain aspects of the type name influences the structure of the corresponding identifier name and how we can use that relationship to improve the identifier. Type inference work like those above do indicate the presence of type clues within the identifier name or its code context. There is some potential for synergy in their work and ours; their approaches highlight how to find type clues in the source code while some of the data in our study highlights where name suggestions could be used to improve identifier naming practices (e.g., in identifiers with collection and boolean types). Improving these practices might make natural-language-based type inference approaches more accurate and efficient, since identifier name structure will be more consistent and well-defined. In addition, examining the data from these type inference approaches might point out some other ways to improve identifier naming appraisal or suggestions using more source code context to better determine when, for example, to use a verb in a boolean or a plural for a list identifier.

6.3. Software Ontology Creation Using Identifier Names

A lot of work has been done in the area of modeling domain knowledge and word relationships by leveraging identifiers [50, 51, 52, 53, 47]. Abebe and Tonella [50] analyze the effectiveness of information retrieval based techniques for filtering domain concepts and relations from implementation details. They show that fully automated techniques based on keywords or topics have low performance but that a semi-automated approach can significantly improve results. Falleri et al., present a way to automatically construct a wordnet-like [43] identifier network from software. Their model is based on synonymy, hypernymy and hyponymy, which are types of relationships between words. Synonyms are words with similar or equivalent meaning; hyper/hyponyms are words which, relative to one another, have a broader or more narrow domain (e.g., dog is a hyponym of animal, animal is a hypernym of dog). Ratiu and Deissenboeck [52] present a framework for mapping real world concepts to program elements bi-directionally. They use a set of object-oriented properties (e.g., isA, hasA) to map relationships between program elements and string matching to map these elements to external concepts. This extends two prior works of theirs; one paper on a previous version of their meta model [53] and a second paper on linking programs to ontologies [51]. Many of these approaches need to split and analyze words found in an identifier in order to connect these identifiers to a model of program semantics (e.g., class hierarchies). All of these approaches rely on identifiers.

While we do not construct an ontology in this paper, many software word ontologies use meta-data about words to understand the relationship between different words. There is a synergistic relationship between the work we present here and software ontologies since stronger ontologies can help us generate and study grammar patterns effectively (e.g., by increasing our knowledge of word meaning/relationships) and the data used in our paper can help construct stronger software word ontologies by supporting research in part of speech taggers for source code and providing annotated word datasets from which word relationships can be deduced. In the future, we aim to use the identifiers and grammar patterns in our dataset to improve software word ontologies.

6.4. Identifier Structure and Semantics Analysis

Liblit et al. [32] discusses naming in several programming languages and makes observations about how natural language influences the use of words in these languages. Schankin et al. [6] focus on investigating the impact of more informative identifiers on code comprehension. Their findings show an advantage of descriptive identifiers over non-descriptive ones. Hofmeister et al [8] compared comprehension of identifiers containing words against identifiers containing letters and/or abbreviations. Their results show that when identifiers contained only words instead of abbreviations or letters, developer comprehension speed increased by 19% on average. Lawrie et al [7] did a study and used three different ”levels” of identifiers. The results show that full word identifiers lead to the best comprehension compared to the other levels studied. Butler's work [9] extends their previous work on Java class identifiers [54] to show that flawed method identifiers are also associated with low-quality code according to static analysis-based metrics. These papers primarily study the words found in identifiers and how they relate to code behavior or comprehension rather than word metadata (e.g., part-of-speech).

Caprile and Tonella [55] analyzes the syntax and semantics of function identifiers. They create classes which can be used to understand the behavior of a function; grouping function identifiers by leveraging the words within them to understand some of the semantics of those identifiers. While they do not identify particular grammar patterns, this study does identify grammatical elements in function identifiers, such as noun and verb, and discusses different roles that they play in expressing behavior both independently and in conjunction using the classes they propose. They also use the classes identified in this prior work to propose methods for restructuring program identifiers [56]. Fry and Shepherd [57, 58] study verb-direct objects to link verbs to the natural-language-representation of the entity they act upon in order to assist in locating action-oriented concerns. The primary concern in this work is identifying the entity (e.g., an object) which a verb is targeting (e.g., the action part of a method name). We study a broader set of identifiers and focus on grammar patterns in general, rather than those containing (or related to) a verb. The grammar structures we prevent may help in identifying verb-direct objects.

Høst and Østvold study method names as part of a line of work discussed in Høst's disseration [59]. This line of work starts by analyzing a corpus of Java method implementations to establish the meanings of verbs in method names based on method behavior, which they measure using a set of attributes which they define [60]. They automatically create a lexicon of verbs that are commonly used by developers and a way to compare verbs in this lexicon by analyzing their program semantics. They build on this work in [61] by using full method names which they refer to as phrases and augment their semantic model by considering a richer set of attributes. The outcome of this work is that they were able to aggregate methods by their phrases and come up with the semantics behind those phrases using their semantic model, therefore modeling the relationship between method names and method behavior. The phrases they discuss are very similar to the grammar patterns we present. They extend this use of phrases by presenting an approach to debug method names [20]. In this work, they designed automated naming rules using method signature elements. They use the phrase refinement from their prior paper, which takes a sequence of part-of-speech tags (i.e., phrases) and concretes them by substituting real words. (e.g., the phrase <verb>-<adjective> might refine to is-empty). They connect these patterns to different method behaviors and use this to determine when a method's name and implementation do not match. They consider this a naming bug. Finally, in [62], Høst and Østvold analyzed how ambiguous verbs in method names makes comprehension of Java programs more difficult. They proposed a way to detect when two or more verbs are synonymous and being used to describe the same behavior in a program; hoping to eliminate these redundancies as well as increase naming consistency and correctness. They perform this detection using two metrics which they introduce called nominal and semantic entropy. Høst and Østvold's work focuses heavily on method naming patterns; connecting these to the implementation of the method to both understand and critique method naming. We focus on a larger variety of identifiers (i.e., not only method names) and have a different goal– specifically exploring and analyzing the wide variety of grammar patterns in different types of identifiers and understanding the effectiveness of part-of-speech taggers. Our work can complement Høst and Østvold's.

Butler [30] studied class identifier names and lexical inheritance; analyzing the effect that interfaces or inheritance has on the name of a given class. For example, a class may inherit from a super class or implement a particular interface. Sometimes this class will incorporate words from the interface name or inherited class in its name. His study builds on work by Singer and Kirkham [63], who identified a grammar pattern for class names of (adjective)* (noun)+ and studies how class names correlate with micro patterns. Amongst Butler's findings, he identifies a number of grammar patterns for class names: (noun)+, (adjective)+ (noun)+, (noun)+ (adjective)+ (noun)+, (verb) (noun)+ and extends these patterns to identify where inherited names and interface names appear in the pattern. The same author also studies Java field, argument, and variable naming structures [31]. Among other results, they identify noun phrases as the most common pattern for field, argument, and variable names. Verb phrases are the second most common. Further, they discuss phrase structures for boolean variables; finding an increase in verb phrases compared to non-boolean variables.

The prior two papers by Butler and Singer focus on class names [63, 30]. Singer primarily identifies the noun phrase grammar pattern so that they can use it to match class names and study micro patterns (i.e., design patterns). Butler focuses more heavily on the class name grammar patterns, with the difference between his work and ours being the focus. Butler focuses on identifying how patterns in a class name repeat in super classes or interfaces, which limits the number of patterns they discuss. We focus on more types of identifiers (i.e., not just class names) and include a larger variation of class name patterns since we are not focusing on on pattern use in super classes/interfaces. In later work [31], Butler adds fields, arguments, and variables. Both papers observe the use of adjectives before nouns but do not discuss noun-adjuncts. The noun phrases in Butler's work make no distinction between nouns and noun-adjuncts; they are all annotated as nouns. We argue, and show, that there is a distinction between them; noun-adjuncts play an important supporting role in the comprehension of head-nouns within noun phrases and should be annotated to reflect this fact for future tools to leverage that metadata. Further, the latter paper [31] discusses patterns at the phrasal level and does not show or discuss grammar patterns at the level of granularity that our work does. This allows us to show and study these patterns more specifically to understand the diversity in pattern structure as well as common, and divergent pattern structures.

Olney [36] also compared taggers for accuracy on identifiers, but only on Java method names which were curated to remove ambiguous words (e.g., abbreviations). Gupta et al [23] designed Posse; a part-of-speech tagger built to annotated source code. While this paper does not study grammar patterns explicitly, they do discuss some grammatical observations leveraged by Posse to help increase its accuracy on source code. They separate these into two categories: method name observations, and class/attribute name observations. For method names, they observe, among other things (we elide to save space): 1) Function names beginning with the base form of a verb sometimes lack the entity/object that is the target of the action. 2) Some function names have an implicit action (e.g., elementAt has an implicit *get* action). For classes, attributes, they observe that noun phrases are typical but note that booleans are frequently exceptions. Our work is primarily complimentary to theirs, as we have shown weaknesses in Posse's tagging algorithm and discuss (in the next Section) some ways to solve these problems.

Binkley et al [22] study grammar patterns for attribute names in classes. They come up with four rules for how to write attribute names: 1) Non-boolean field names should not contain a present tense verb, 2) field names should never only be a verb, 3) field names should never only be an adjective, and 4) boolean field names should contain a third person form of the verb “to be” or the auxiliary verb “should”. This work focuses primarily on attributes; we deal with a larger variety of identifiers and discuss a larger range of grammar patterns. Our data confirms some of Binkley’s findings, particularly in the use of verbs in boolean field names.

7. Discussion

This study has implications for both the study of identifier names and the improvement of part-of-speech tagging techniques applied to identifiers. Below, we discuss how our results augment researchers’ understanding of identifier names and how these results can be applied to improve part-of-speech taggers for identifiers.

As a general takeaway, **the implementation is a very important resource for annotating identifier names with the correct part-of-speech**, as we will discuss more in the takeaways below. Words in code change meaning based on context just like in English, but unlike English, that context is not always in natural language; sometimes, the context is the behavior specified by code. One example from above is the identifier *do forward*, which we gave a grammar pattern of *V V* due to the fact that it is a method that redirects an entity which is not present in the identifier name (i.e., the user) when it is called. However, if we change the method’s implementation so that instead of being used as a synonym for *redirect*, *forward* refers to (for example) moving a video game sprite forward on a screen, then we might give it the grammar pattern *V VM* (though *move* might be a better verb than *do* in this case). This also means that domain knowledge and code analysis may be critical resources to correct part-of-speech tagging in some situations.

7.1. Takeaways from RQ1

Noun-adjuncts are ubiquitous in identifier names: Noun phrases are the most common grammar pattern outside of functions, where verb phrases are more common. However, unlike noun and verb phrases in English, the adjectives in these identifiers tend to be *noun adjuncts* (i.e., noun modifiers); nouns which behave like adjectives. This fact highlights how developers use noun-words within the domain of the software, or computer science in general, as adjectives to modify the meaning of the entity represented by the head-noun, which is typically the last (i.e., rightmost) word in the identifier. The ubiquity of noun-adjuncts has important implications for part-of-speech techniques, especially with respect to unknown word handling, as shown by Swum's accuracy. Noun-adjunct should be the default unknown word annotation for certain contexts–specifically words at the beginning or middle of declaration-statement, attribute, or parameter variables that have non-collection and non-boolean types. With the exception of Swum, none of the part-of-speech taggers we studied are effective at detecting noun adjuncts or, as a consequence, head-nouns.

Likewise, other tools which use part-of-speech to perform analysis should at least be aware of the relationship between noun modifiers and head-nouns, as well as their ubiquity, especially when leveraging a tagger which does not accurately recognize them. Otherwise, they will potentially misunderstand which word is the head-noun; it is important to distinguish the head-noun from adjectives and noun-adjuncts which specify it. The ubiquity of noun-adjuncts within noun phrases has also not been discussed or quantitatively confirmed in research prior to this paper. Prior work has discussed noun phrases and compound words. Butler [30, 31] and Deissenboeck and Pizka [5] observe the use of adjectives before nouns but do not discuss noun-adjuncts. The noun phrases in both studies make no distinction between nouns and noun-adjuncts; they are all annotated as nouns. However, Deissenboeck and Pizka do mention the use of compound words in programming which follow a *modifier head-noun* pattern similar to noun modifiers. We argue, and show, that there is a distinction between nouns and noun-adjuncts; noun-adjuncts play an important role in the comprehension of head-nouns and should be annotated to reflect this role such that the difference between them is plainly obvious to tools and researchers that leverage POS annotations. Gupta [23], who created Posse, made the same noun-adjunct observation as we do (i.e., they annotate using NM), but give no data or discussion about their frequency.

Several identifier characteristics have an effect on part-of-speech annotation: Function identifiers are more likely to contain a verb and be represented by a verb phrase. Attribute, parameter, and declaration-statement identifiers typically have singular noun-phrase grammar patterns **unless** they have a type that is a collection or Boolean. Collection type identifiers have somewhat increased probability of using a plural head-noun, causing them to be plural noun phrases. This observation does not hold for function identifiers with collection return types because plural function names are more likely to be a reference to the multiplicity of the entity (e.g., an object) being operated on by the function. Boolean type identifiers are more likely to contain a verb. These patterns are opportunities for researchers to recommend when developers should be using plurals and verbs in their identifier names. These could initially be simple reminders; asking developers to consider using a verb or a plural and giving them the option of ignoring or accepting the recommendation. As research more firmly grasps the situations in which these practices should be followed, suggestions can be strengthened. This would not be very invasive, and might help increase consistency in naming practices. The characteristic of Boolean-type identifiers containing verbs has been observed before by Høst and Østvold, Binkley, and Gupta [22, 23, 20] and quantified by Butler [31]; our work helps confirm their results by considering a larger number of identifiers from a broader range of categories (function, attribute names, etc).

7.2. Takeaways from RQ2

Part-of-speech taggers still require significant improvements to be effective on identifiers, and may be augmented by combining their individual strengths. Preambles are a significant problem for all taggers: Our results show that even the best tagger for identifiers has an accuracy of between 50.2% and 67.8%. The other two taggers had much lower accuracy at the grammar pattern level. At the part-of-speech tag level, Stanford was the most accurate tagger; able to annotate not only a larger set of part-of-speech tags than Swum or Posse, but also more accurately than both in many cases. The contrast in performance between the taggers at different levels (i.e., grammar-pattern level and part-of-speech level) indicates that these approaches are likely complimentary; their combined output can help us find the correct grammar pattern. Some of these are straightforward. Noun plurals, for example, are not annotated by Swum or Posse. Stanford has 71.85% accuracy on them and could inform us when Swum or Posse has missed a plural. Stanford is also the most effective at annotating digits, having agreed with them human annotations 100% of the time; the second-best tagger for digits is Swum at 18.52% accuracy. Stanford is also the only tagger that annotates conjunctions somewhat correctly (although, there were not many in the data set), while Posse and Stanford are more accurate at detecting prepositions than Swum; Swum has 29.79% accuracy on prepositions while Posse and Stanford have 62.77% and 90.43% accuracy respectively. On the other end, Swum is far more effective than Stanford and Posse at correctly annotating noun modifiers (noun-adjuncts) with an accuracy of 94.01% versus Stanford's 15.71% and Posse's 23.25%.

All in all, the strengths of one tagger are the weaknesses of the others. We can take advantage of the strengths and weaknesses of each tagger to produce better part-of-speech tagging annotations. Increasing overall accuracy on plural words and digits by combining Swum and Stanford's output is very simple and will cause immediate improvements in part-of-speech annotations. Increasing accuracy on other part-of-speech annotations will require more sophistication, but our data indicates that ensemble tagging [64] may be an effective approach. Further, all taggers have significant problems detecting preambles due to fact that preamble detection requires taggers to recognize naming conventions and domain information. Future research should focus on building naming convention recognition into taggers more effectively and finding ways to determine when a preamble is being used as a namespace. Frequency and position within the identifier might be useful in this regard, but will likely lead to false positives. It may be that domain terms and abbreviations will need to be identified as a configuration step before tagging. To our knowledge, this is the first formal comparison of tagger output and accuracy which has pinpointed specific areas of tagger synergy for identifiers in source code across multiple types of identifiers. Olney performed a comparison of taggers, but only on method names [36] and they do not give an in-depth discussion of individual tagger weaknesses on either individual parts of speech or on grammar patterns.

Certain identifier configurations can improve the output of off-the-shelf taggers: Stanford+I is a technique where one adds an *I* before a method name to help Stanford identify verbs more effectively. Our results confirm this– prepending *I* before method names increased the accuracy of Stanford by around 3%. In addition, we tried two ways of feeding data to Stanford. Typically, Stanford reads sentences and paragraphs from a text file. We tried both feeding 1 identifier per file (i.e., 267 files, each with 1 identifier) into Stanford and multiple identifiers in a single file (i.e., 1 file with 267 identifiers) separated by a period and a new line. Stanford's grammar pattern level accuracy did not change under either of these configurations, although it did make different tagging decisions in a small minority of identifiers. As these changes did not affect its grammar pattern level accuracy, and made minor changes overall, we did not report numbers. However, we do note that the way identifiers are fed to Stanford will have some small impact on the part of speech annotation it uses for some words. The Stanford+I technique has been used in prior research [36]. In this paper, we confirm that it improves Stanford's tagging on function names.

Certain verb conjugations are frequently used as adjectives in certain types of identifiers: Past tense verb (VBD), present participle verb (VBG), and past participle verb (VBN) are all used as adjectives in many situations within our data set. For example, *sortedIndicesBuf*, *waitingList*, and *adjustedGradient* where *sorted* is a past tense verb (VBD), *waiting* is a present participle (VBG), and *adjusted* is a past participle verb (VBN). While these are all verbs, they are used to describe the head-noun in their respective identifiers and should therefore be annotated using *NM*. One particular situation where this happened a lot is when there was a non-function variable referencing a domain of functions. For example, *get Protocol Method* is a parameter which holds a reference to a method, which can be one of many different methods passed in through the parameter. In other words, *get protocol method* is a noun phrase with the grammar pattern *NM NM N*, where the words *get* and *protocol* describe the type of method represented by this parameter instead of the action of a get-method. Where *get* would typically be a verb, it is instead a noun modifier. We find that these verb-adjectives are typically found in attribute, parameter, and declaration-statement identifiers, meaning that verbs in these situations need to be more highly scrutinized than when they are found in functions. Detecting when an identifier represents a function, event, or generally any function callback could significantly help detect verb-adjectives. In addition, verbs that are of one of the three forms above and are found in attributes, declaration-statements, or parameters are more likely to be noun modifiers than they are to be verbs, as shown by the fact that Stanford's output improved when we interpreted them as noun modifiers instead of verbs (Table 11). This can be directly applied to improve off-the-shelf tagger accuracy and, as a result, any tagging which relies on the output of multiple taggers since Swum and Posse, for example, do not use these conjugated verb annotations and are less able to determine when a verb should be an adjective. This result also means that part-of-speech taggers for source code should support these conjugated verb forms.

Posse and Swum's developers both recognized the difficulty of tagging verb conjugations [23, 34]; Høst and Østvold also observes this for method names [20]. We provide data on this problem that the prior three papers do not include. Specifically our data set includes verb conjugations in a larger sample of manually validated grammar patterns that can be used for further analysis and tagger training; we discuss one way to help determine how to interpret verb conjugations in a source code context (i.e., use verb for function and noun-adjunct for other categories); and we show that this interpretation does matter, since Stanford's accuracy varies with how we interpret verb conjugations.

7.3. Takeaways from RQ3

Many grammar patterns in our data set were infrequent, but still appeared to represent specific forms of behavior: Grammar patterns that contain prepositional phrases and determiners may appear more frequently in software following specific design patterns or software architectures. For example, prepositional phrase grammar patterns like *P NM* N* or *V P* correlate with event-driven-programming functionality with identifiers such as *onclick()* and *onKeypress()*; conversion functionality such as *ToString()* and *ToInt()*; or common data analysis operations such as *order by* and *group by*. The same holds true for patterns containing determiners. Identifiers such as *all action roots*, *all open indices*, and *all invocation matchers* should be more closely studied to understand more about their usage contexts. While prepositional phrase and determiner patterns were not common in our data set, they were used in several systems independently. Some of these patterns are more typical in certain domains. Prepositional phrase grammar patterns are used in event-driven programming very frequently, for example. The patterns presented in RQ3 represent future directions for research; there are not enough examples of the patterns we presented in this research question to draw strong conclusions, but their existence is evidence that these grammar patterns are common in other contexts (e.g., different architecture and design patterns). We argue that studying grammar patterns in these other contexts may result in more examples or even new patterns. These domain-specific patterns would then be very useful for suggesting and appraising identifier names within those contexts.

7.4. Takeaways from RQ4

The most common C/C++ and Java grammar patterns are very similar, but they differ in their usage of full words and abbreviations in identifiers: We observed a difference in the rate at which C/C++ and Java use abbreviations in their identifiers. This difference may make it more difficult for part-of-speech taggers, and other natural language approaches, to analyze C/C++ systems compared to Java systems. In addition, our results suggest that if we only control for identifier category and programming language, then use of Java or C/C++ does not have a significant impact on common identifier grammar patterns for systems written in these languages. However, there are a number of other confounding factors which future research should explore in order to more definitively ascertain the difference (or lack thereof) between grammar patterns found in different programming languages. Some factors which we recommend future research control for are: architecture patterns, design patterns, problem domain, programming paradigm, and modernity; some of our data (e.g., RQ3) suggests that these may have an influence on the grammar patterns identifiers follow.

8. Conclusions and Future Work

We have identified and discussed a frequent, specific, and diverse, set of grammar patterns which developers use to express program semantics and behavior. These patterns are obtained directly from observations made on source code and provide researchers with a glimpse into how different grammar patterns are used to express different forms of program semantics. We do not argue that we have discovered all grammar patterns. On the contrary, our discussion from RQ3 suggests that there are more to be discovered. The grammar patterns presented in this study are a starting point for the creation of a taxonomy which should be curated over time and through further research. The dataset that we used to perform our comparison contains a fine grain set of grammar patterns from the systems we studied, which is a more granular view of these patterns than presented in prior work. This dataset is available on our webpage [8] to help improve POS taggers for software and encourage replication.

Our direct future work includes: 1) improving POS taggers for source code by leveraging ensemble tagging [64], 2) using grammar patterns and static analysis models [25, 65, 26] to appraise identifier names, and 3) studying grammar

[8] https://scanl.org/

patterns in specific design/architectural patterns and on test code. The results from this work will be particularly useful for improving the analysis of words, and their relationship to other nearby words, in the lexicon of a given software system. In particular, our observations will allow for better recognition of head-nouns (or lack thereof), verb-direct object pairs, and phrases used by code search and modeling tools like Swum and Sando [66, 34] and provide support for AI/ML techniques which recommend names [14, 15]; allowing them to more effectively choose words, even neologisms, that fit an appropriate pattern. The grammar patterns we have identified, and future extension to the collection, will provide greater understanding of the role different patterns play in comprehension and generation of identifier names.

9. ACKNOWLEDGEMENT

This material is based upon work supported by the National Science Foundation under Grant No. 1850412.

References

[1] T. A. Corbi, Program understanding: Challenge for the 1990s, IBM Systems Journal 28 (2) (1989) 294–306. doi:10.1147/sj.282.0294.

[2] R. C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship, 1st Edition, Prentice Hall PTR, Upper Saddle River, NJ, USA, 2008.

[3] A. von Mayrhauser, A. M. Vans, Program understanding behavior during debugging of large scale software, in: Papers Presented at the Seventh Workshop on Empirical Studies of Programmers, ESP '97, ACM, New York, NY, USA, 1997, pp. 157–179. doi:10.1145/266399.266414.
URL http://doi.acm.org/10.1145/266399.266414

[4] M. Fisher, A. Cox, L. Zhao, Using sex differences to link spatial cognition and program comprehension, in: Proceedings of the 22Nd IEEE International Conference on Software Maintenance, ICSM '06, IEEE Computer Society, Washington, DC, USA, 2006, pp. 289–298. doi:10.1109/ICSM.2006.72.
URL https://doi.org/10.1109/ICSM.2006.72

[5] F. Deissenboeck, M. Pizka, Concise and consistent naming, Software Quality Journal 14 (3) (2006) 261–282. doi:10.1007/s11219-006-9219-1.
URL https://doi.org/10.1007/s11219-006-9219-1

[6] A. Schankin, A. Berger, D. V. Holt, J. C. Hofmeister, T. Riedel, M. Beigl, Descriptive compound identifier names improve source code comprehension, in: Proceedings of the 26th Conference on Program Comprehension, ICPC '18, ACM, New York, NY, USA, 2018, pp. 31–40. doi:10.1145/3196321.3196332.
URL http://doi.acm.org/10.1145/3196321.3196332

[7] D. Lawrie, C. Morrell, H. Feild, D. Binkley, What's in a name? a study of identifiers, in: 14th IEEE International Conference on Program Comprehension (ICPC'06), 2006, pp. 3–12. doi:10.1109/ICPC.2006.51.

[8] J. Hofmeister, J. Siegmund, D. V. Holt, Shorter identifier names take longer to comprehend, in: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2017, pp. 217–227. doi:10.1109/SANER.2017.7884623.

[9] S. Butler, M. Wermelinger, Y. Yu, H. Sharp, Exploring the influence of identifier names on code quality: An empirical study, in: Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on, IEEE, 2010, pp. 156–165.

[10] A. A. Takang, P. A. Grubb, R. D. Macredie, The effects of comments and identifier names on program comprehensibility: an experimental investigation, J. Prog. Lang. 4 (1996) 143–167.

[11] D. Binkley, D. Lawrie, C. Morrell, The need for software specific natural language techniques, Empirical Softw. Engg. 23 (4) (2018) 2398–2425. doi:10.1007/s10664-017-9566-5.
URL https://doi.org/10.1007/s10664-017-9566-5

[12] C. D. Newman, M. J. Decker, R. S. AlSuhaibani, A. Peruma, D. Kaushik, E. Hill, An empirical study of abbreviations and expansions in software artifacts, in: Proceedings of the 35th IEEE International Conference on Software Maintenance, IEEE, 2019.

[13] Y. Kashiwabara, Y. Onizuka, T. Ishio, Y. Hayase, T. Yamamoto, K. Inoue, Recommending verbs for rename method using association rule mining, in: 2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE), 2014, pp. 323–327. doi:10.1109/CSMR-WCRE.2014.6747186.

[14] M. Allamanis, E. T. Barr, C. Bird, C. Sutton, Suggesting accurate method and class names, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, ACM, New York, NY, USA, 2015, pp. 38–49. doi:10.1145/2786805.2786849.
URL http://doi.acm.org/10.1145/2786805.2786849

[15] K. Liu, D. Kim, T. F. Bissyandé, T. Kim, K. Kim, A. Koyuncu, S. Kim, Y. Le Traon, Learning to spot and refactor inconsistent method names, in: Proceedings of the 40th International Conference on Software Engineering, ICSE 2019, ACM, New York, NY, USA, 2019.

[16] S. L. Abebe, P. Tonella, Automated identifier completion and replacement, in: 2013 17th European Conference on Software Maintenance and Reengineering, 2013, pp. 263–272. doi:10.1109/CSMR.2013.35.

[17] V. Arnaoudova, M. Di Penta, G. Antoniol, Y. Guéhéneuc, A new family of software anti-patterns: Linguistic anti-patterns, in: 2013 17th European Conference on Software Maintenance and Reengineering, 2013, pp. 187–196. doi:10.1109/CSMR.2013.28.

[18] V. Arnaoudova, L. M. Eshkevari, M. D. Penta, R. Oliveto, G. Antoniol, Y.-G. Gueheneuc, Repent: Analyzing the nature of identifier renamings, IEEE Trans. Softw. Eng. 40 (5) (2014) 502–532. doi:10.1109/TSE.2014.2312942.
URL https://doi.org/10.1109/TSE.2014.2312942

[19] A. Peruma, M. W. Mkaouer, M. J. Decker, C. D. Newman, Contextualizing rename decisions using refactorings and commit messages, in: Proceedings of the 19th IEEE International Working Conference on Source Code Analysis and Manipulation, IEEE, 2019.

[20] E. W. Høst, B. M. Østvold, Debugging method names, in: Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming, Genoa, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 294–317. doi:10.1007/978-3-642-03013-0_14.
URL http://dx.doi.org/10.1007/978-3-642-03013-0_14

[21] A. Peruma, M. W. Mkaouer, M. J. Decker, C. D. Newman, An empirical investigation of how and why developers rename identifiers, in: International Workshop on Refactoring 2018, 2018. doi:10.1145/3242163.3242169.
URL http://doi.acm.org/10.1145/3242163.3242169

[22] D. Binkley, M. Hearn, D. Lawrie, Improving identifier informativeness using part of speech information, in: Proceedings of the 8th Working Conference on Mining Software Repositories, MSR '11, ACM, New York, NY, USA, 2011, pp. 203–206. doi:10.1145/1985441.1985471.
URL http://doi.acm.org/10.1145/1985441.1985471

[23] S. Gupta, S. Malik, L. Pollock, K. Vijay-Shanker, Part-of-speech tagging of program identifiers for improved text-based software engineering tools, in: 2013 21st International Conference on Program Comprehension (ICPC), 2013, pp. 3–12. doi:10.1109/ICPC.2013.6613828.

[24] E. Hill, L. Pollock, K. Vijay-Shanker, Improving source code search with natural language phrasal representations of method signatures, in: 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), 2011, pp. 524–527. doi:10.1109/ASE.2011.6100115.

[25] N. Dragan, M. L. Collard, J. I. Maletic, Reverse engineering method stereotypes, in: Proceedings of the 22Nd IEEE International Conference on Software Maintenance, ICSM '06, IEEE Computer Society, Washington, DC, USA, 2006, pp. 24–34. doi:10.1109/ICSM.2006.54.
URL http://dx.doi.org/10.1109/ICSM.2006.54

[26] R. S. Alsuhaibani, C. D. Newman, M. L. Collard, J. I. Maletic, Heuristic-based part-of-speech tagging of source code identifiers and comments, in: 2015 IEEE 5th Workshop on Mining Unstructured Data (MUD), 2015, pp. 1–6. doi:10.1109/MUD.2015.7327960.

[27] C. D. Newman, R. S. AlSuhaibani, M. L. Collard, J. I. Maletic, Lexical categories for source code identifiers, in: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2017, pp. 228–239. doi:10.1109/SANER.2017.7884624.

[28] R. Jongeling, P. Sarkar, S. Datta, A. Serebrenik, On negative results when using sentiment analysis tools for software engineering research, Empirical Software Engineering (01 2017). doi:10.1007/s10664-016-9493-x.

[29] E. Hill, L. Pollock, K. Vijay-Shanker, Automatically capturing source code context of nl-queries for software maintenance and reuse, in: 2009 IEEE 31st International Conference on Software Engineering, 2009, pp. 232–242. doi:10.1109/ICSE.2009.5070524.

[30] S. Butler, M. Wermelinger, Y. Yu, H. Sharp, Mining java class naming conventions, in: 2011 27th IEEE International Conference on Software Maintenance (ICSM), 2011, pp. 93–102. doi:10.1109/ICSM.2011.6080776.

[31] S. Butler, M. Wermelinger, Y. Yu, A survey of the forms of java reference names, in: 2015 IEEE 23rd International Conference on Program Comprehension, 2015, pp. 196–206. doi:10.1109/ICPC.2015.30.

[32] B. Liblit, A. Begel, E. Sweetser, Cognitive perspectives on the role of naming in computer programs, in: In Proc. of the 18th Annual Psychology of Programming Workshop, 2006.

[33] M. Hucka, Spiral: splitters for identifiers in source code files, Journal of Open Source Software 3 (2018) 653. doi:10.21105/joss.00653.

[34] E. Hill, Integrating natural language and program structure information to improve software search and exploration, Ph.D. thesis, Newark, DE, USA, aAI3423409 (2010).

[35] K. Toutanova, C. D. Manning, Enriching the knowledge sources used in a maximum entropy part-of-speech tagger, in: Proceedings of the 2000 Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora: Held in Conjunction with the 38th Annual Meeting of the Association for Computational Linguistics - Volume 13, EMNLP '00, Association for Computational Linguistics, Stroudsburg, PA, USA, 2000, pp. 63–70. doi:10.3115/1117794.1117802.
URL https://doi.org/10.3115/1117794.1117802

[36] W. Olney, E. Hill, C. Thurber, B. Lemma, Part of speech tagging java method names, in: 2016 IEEE International Conference on Software Main-

tenance and Evolution (ICSME), 2016, pp. 483–487. doi:10.1109/ICSME.2016.80.

[37] E. Hill, Z. P. Fry, H. Boyd, G. Sridhara, Y. Novikova, L. Pollock, K. Vijay-Shanker, Amap: Automatically mining abbreviation expansions in programs to enhance software maintenance tools, in: Proceedings of the 2008 International Working Conference on Mining Software Repositories, MSR '08, Association for Computing Machinery, New York, NY, USA, 2008, p. 79–88. doi:10.1145/1370750.1370771.
URL https://doi.org/10.1145/1370750.1370771

[38] C. Simonyi, M. Heller, The hungarian revolution, BYTE 16 (8) (1991) 131–ff.

[39] S. L. Abebe, P. Tonella, Natural language parsing of program element names for concept extraction, in: Proceedings of the 2010 IEEE 18th International Conference on Program Comprehension, ICPC '10, IEEE Computer Society, USA, 2010, p. 156–159. doi:10.1109/ICPC.2010.29.
URL https://doi.org/10.1109/ICPC.2010.29

[40] C. D. Manning, H. Schütze, Foundations of Statistical Natural Language Processing, MIT Press, Cambridge, MA, USA, 1999.

[41] M. L. Collard, J. I. Maletic, srcml 1.0: Explore, analyze, and manipulate source code, in: 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2016, pp. 649–649. doi:10.1109/ICSME.2016.36.

[42] S. W. Ambler, A. Vermeulen, G. Bumgardner, The Elements of Java Style, Cambridge University Press, USA, 1999.

[43] G. A. Miller, Wordnet: a lexical database for english, Communications of the ACM 38 (11) (1995) 39–41.

[44] H. Liu, Q. Liu, Y. Liu, Z. Wang, Identifying renaming opportunities by expanding conducted rename refactorings, IEEE Transactions on Software Engineering 41 (9) (2015) 887–900.

[45] H. Liu, Q. Liu, C.-A. Staicu, M. Pradel, Y. Luo, Nomen est omen: Exploring and exploiting similarities between argument and parameter names, in: Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on, IEEE, 2016, pp. 1063–1073.

[46] A. Peruma, M. W. Mkaouer, M. J. Decker, C. D. Newman, Contextualizing rename decisions using refactorings, commit messages, and data types, Journal of Systems and Software 169 (2020) 110704. doi:https://doi.org/10.1016/j.jss.2020.110704.
URL http://www.sciencedirect.com/science/article/pii/S0164121220301503

[47] J.-R. Falleri, M. Huchard, M. Lafourcade, C. Nebut, V. Prince, M. Dao, Automatic extraction of a wordnet-like identifier network from software, in: Proceedings of the 2010 IEEE 18th International Conference on Program Comprehension, ICPC '10, IEEE Computer Society, USA, 2010, p. 4–13. doi:10.1109/ICPC.2010.12.
URL https://doi.org/10.1109/ICPC.2010.12

[48] R. S. Malik, J. Patra, M. Pradel, Nl2type: Inferring javascript function types from natural language information, in: Proceedings of the 41st International Conference on Software Engineering, ICSE '19, IEEE Press, 2019, p. 304–315. doi:10.1109/ICSE.2019.00045.
URL https://doi.org/10.1109/ICSE.2019.00045

[49] V. J. Hellendoorn, C. Bird, E. T. Barr, M. Allamanis, Deep learning type inference, in: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2018, Association for Computing Machinery, New York, NY, USA, 2018, p. 152–162. doi:

10.1145/3236024.3236051.
URL https://doi.org/10.1145/3236024.3236051

[50] S. L. Abebe, P. Tonella, Towards the extraction of domain concepts from the identifiers, in: Proceedings of the 2011 18th Working Conference on Reverse Engineering, WCRE '11, IEEE Computer Society, USA, 2011, p. 77–86. doi:10.1109/WCRE.2011.19.
URL https://doi.org/10.1109/WCRE.2011.19

[51] D. Ratiu, F. Deissenboeck, Programs are knowledge bases, Vol. 2006, 2006, pp. 79 – 83. doi:10.1109/ICPC.2006.41.

[52] D. Ratiu, F. Deissenboeck, From reality to programs and (not quite) back again, in: Proceedings of the 15th IEEE International Conference on Program Comprehension, ICPC '07, IEEE Computer Society, USA, 2007, p. 91–102. doi:10.1109/ICPC.2007.22.
URL https://doi.org/10.1109/ICPC.2007.22

[53] F. Deissenboeck, D. Ratiu, A unified meta-model for concept-based reverse engineering, in: In Proceedings of the 3rd International Workshop on Metamodels, Schemas, Grammars and Ontologies (ATEM'06, 2006.

[54] S. Butler, M. Wermelinger, Y. Yu, H. Sharp, Relating identifier naming flaws and code quality: An empirical study, in: 2009 16th Working Conference on Reverse Engineering, 2009, pp. 31–35. doi:10.1109/WCRE.2009.50.

[55] C. Caprile, P. Tonella, Nomen est omen: analyzing the language of function identifiers, in: Sixth Working Conference on Reverse Engineering (Cat. No.PR00303), 1999, pp. 112–122. doi:10.1109/WCRE.1999.806952.

[56] Caprile, Tonella, Restructuring program identifier names, in: Proceedings 2000 International Conference on Software Maintenance, 2000, pp. 97–107. doi:10.1109/ICSM.2000.883022.

[57] D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, K. Vijay-Shanker, Using natural language program analysis to locate and understand action-oriented concerns, in: Proceedings of the 6th International Conference on Aspect-oriented Software Development, AOSD '07, ACM, New York, NY, USA, 2007, pp. 212–224. `doi:10.1145/1218563.1218587`.
URL `http://doi.acm.org/10.1145/1218563.1218587`

[58] Z. P. Fry, D. Shepherd, E. Hill, L. Pollock, K. Vijay-Shanker, Analysing source code: looking for useful verb-direct object pairs in all the right places, IET Software 2 (1) (2008) 27–36. `doi:10.1049/iet-sen:20070112`.

[59] E. W. Høst, Meaningful method names, 2011.

[60] E. Host, B. Ostvold, The programmer's lexicon, volume i: The verbs, 2007, pp. 193 – 202. `doi:10.1109/SCAM.2007.18`.

[61] E. W. Høst, B. M. Østvold, The java programmer's phrase book, in: D. Gašević, R. Lämmel, E. Van Wyk (Eds.), Software Language Engineering, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 322–341.

[62] E. W. Høst, B. M. Østvold, Canonical method names for java, in: B. Malloy, S. Staab, M. van den Brand (Eds.), Software Language Engineering, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 226–245.

[63] J. Singer, C. Kirkham, Exploiting the correspondence between micro patterns and class names, in: 2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation, 2008, pp. 67–76. `doi:10.1109/SCAM.2008.23`.

[64] F. Dell'Orletta, Ensemble system for part-of-speech tagging, in: Proceedings of EVALITA 2009, 2009.

[65] Y. Gil, O. Marcovitch, M. Orrú, A nano-pattern language for java, Journal of Computer Languages 54 (2019) 100905. `doi:https://doi.org/10.1016/j.cola.2019.100905`.

URL http://www.sciencedirect.com/science/article/pii/S1045926X18302453

[66] D. Shepherd, K. Damevski, B. Ropski, T. Fritz, Sando: An extensible local code search framework, in: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE '12, Association for Computing Machinery, New York, NY, USA, 2012. doi:10.1145/2393596.2393612.
URL https://doi.org/10.1145/2393596.2393612