Empirical Software Engineering manuscript No.
(will be inserted by the editor)

On the Structure and Semantics of Identifier Names Containing Closed Syntactic Category Words

Christian D. Newman · Anthony Peruma · Eman Abdullah AlOmar · Mahie Crabbe · Syreen Banabilah · Reem S. Alsuhaibani · Michael J. Decker · Farhad Akhbardeh · Marcos Zampieri · Mohamed Wiem Mkaouer · Jonathan I. Maletic

the date of receipt and acceptance should be inserted later

Abstract Identifier names are crucial components of code, serving as primary clues for developers to understand program behavior. This paper investigates the linguistic structure of identifier names by extending the concept of grammar patterns, which represent the part-of-speech (PoS) sequences underlying identifier phrases. The specific focus is on closed syntactic categories (e.g., prepositions, conjunctions, determiners), which are rarely studied in software engineering despite their central role in general natural language. To study these categories, the Closed Category Identifier Dataset (CCID), a new manually annotated dataset of 1,275 identifiers drawn from 30 open-source systems, is constructed and pre-

Christian D. Newman
E-mail: cdnvse@rit.edu

Anthony Peruma
E-mail: peruma@hawaii.edu

Eman Abdullah AlOmar
E-mail: ealomar@stevens.edu

Mahie Crabbe
E-mail: mahi3@hawaii.edu

Syreen Banabilah
E-mail: sbanabil@kent.edu

Reem S. AlSuhaibani
E-mail: rsuhaibani@psu.edu.sa

Michael J. Decker
E-mail: mdecke@bgsu.edu

Farhad Akhbardeh
E-mail: farhad.akhbardeh@briarcliff.edu

Marcos Zampieri
E-mail: mzampier@gmu.edu

Mohamed Wiem Mkaouer
E-mail: mmkaouer@umich.edu

Jonathan I. Maletic
E-mail: jmaletic@kent.edu

sented. The relationship between closed-category grammar patterns and program behavior is then analyzed using grounded-theory-inspired coding, statistical, and pattern analysis. The results reveal recurring structures that developers use to express concepts such as control flow, data transformation, temporal reasoning, and other behavioral roles through naming. This work contributes an empirical foundation for understanding how linguistic resources encode behavior in identifier names and supports new directions for research in naming, program comprehension, and education.

Keywords identifier naming · program comprehension · part of speech tagging · software maintenance and evolution · software linguistics · closed category terms · naming conventions

1 Introduction

Developers spend a significant amount of time reading and comprehending code [21, 50], and identifier names play a central role in this process, accounting for roughly 70% of all code characters [23]. Prior work shows that the quality of identifier names significantly impacts comprehension [11, 16, 27, 36, 44, 68, 72], supports tooling [13, 56], and poses persistent pedagogical challenges [30, 74]. These challenges motivate research into how naming practices encode meaning, and how we might better characterize or improve them.

A key obstacle in studying identifier names is measuring the semantics they convey, not just at the level of individual terms, but in the structure and composition of entire names. Some approaches cluster identifiers by terms or embeddings [5, 48], while others analyze them using syntactic or static roles [7, 25, 54]. In this work, we focus instead on **grammar patterns** [55]: sequences of part-of-speech (PoS) tags that abstract the phrasal structure of identifiers. Grammar patterns provide a syntactic lens through which naming semantics can be studied at scale, offering insight into how term combinations convey behavioral meaning.

At a high level, PoS can be split into two Syntactic Categories: **open** and **closed**. Most identifier naming research has focused on **open-category**, which includes nouns and verbs. The set of open category terms evolves and expands (in terms of new words) over time as new domains emerge and evolve. In contrast, **closed-category** (e.g., prepositions, conjunctions, determiners) are drawn from a fixed set and serve functional roles in language; this set of terms rarely sees new words introduced over time. These terms have received little attention in the software literature, despite their importance in human languages. Identifying closed-category terms in code is also nontrivial: for example, the word `and` may represent a conjunction or a logical operator, depending on context, making PoS tagging a prerequisite for meaningful analysis.

The goal of this paper is to investigate how **closed-category terms** are used in identifier names to express program behavior, using the grammar patterns (see Section 3 for definitions) that these terms appear within to provide insights into how these terms interact with the other terms around them. We extend prior research on general grammar patterns [53, 55] by introducing and analyzing the Closed Category Identifier Dataset (CCID), a manually annotated corpus of 1,275 identifiers from 30 open-source systems. Unlike raw term-based approaches, grammar patterns abstract away surface vocabulary, allowing us to characterize naming

conventions by their syntactic structure. By examining both the patterns and the concrete terms that instantiate them, we explore how developers use compact linguistic forms to encode behavioral semantics in code. Specifically, we contribute:

- **A new dataset (CCID)** of identifiers containing closed-category terms, annotated with PoS tags, grammar patterns, and contextual metadata.
- **A mixed-methods analysis** combining grounded-theory-style coding with statistical evaluation to characterize the semantics of closed-category grammar patterns and their constituent terms.
- **An evaluation of how these patterns correlate with programming context, language, and domain**.

Our findings have implications for both human and automated naming support. Grammar patterns provide a structured approach to analyzing naming behavior, identifying potential inconsistencies, and providing naming suggestions. For AI-based tools, they offer scaffolding to align generated names with human conventions. For developers and educators, they reveal naming idioms that can support clearer communication and pedagogy. In this study, we address the following research questions:

RQ1: **What behavioral roles do closed-category terms play in source code identifiers?** To address this question, we conducted a grounded-theory-inspired study on a manually annotated dataset of identifiers containing closed-category terms: prepositions, conjunctions, determiners, and numerals. Through open coding and memoing, we develop axial and selective codes that describe the behavioral functions these terms convey in source code, such as data flow, condition handling, or execution sequencing. This process allows us to uncover not only common grammar patterns but also the communicative intent behind developers' use of closed-category terms. Our goal is to characterize the nuanced and purposeful ways in which these terms encode program behavior and convey information.

RQ2: **How do closed-category terms correlate with structural, programming language, and domain-specific contexts in software?** To answer this question, we quantitatively analyze the distribution of closed-category terms across multiple dimensions: source-code-local structure (e.g., function names, parameters, class names), programming languages (e.g., Java, C++, C), and system domains (e.g., libraries, frameworks, domain-specific applications). We use statistical tests to examine whether these terms appear disproportionately in specific contexts. These correlations help us determine whether developers systematically leverage closed-category terms to express behavior in ways that are shaped by structural conventions, linguistic norms, or domain constraints.

This paper is organized as follows. Section 2 provides our reasoning on why it is essential to study this topic. Section 3 gives background on grammar patterns in the context of identifier names. Section 4 provides a detailed explanation of the methods used for conducting the investigation. Our Evaluations are presented in Sections 5 and 6. Related work on identifier names is in Section 7. Discussion of the results is in section 8, followed by Threats to Validity in Section 9. Conclusions are in Section 10 and Data Availability in Section 11.5.

Table 1: Examples of closed-category grammar patterns

Identifier Example	Grammar Pattern
action to index map	N P NM N
as binary	P N
time for each line	N P DT N
server and port	N CJ N
open if empty	V CJ NM
adjust to camera	V P N

2 Why Study Closed-Category Naming Patterns?

Closed-category terms are relatively uncommon in identifier names. Because they are uncommon, their presence raises an important question: *When developers do use these terms, what specific meaning or behavior are they trying to convey?* We hypothesize that developers include closed-category terms deliberately, as a way to encode behaviorally specific semantics that are lost or obscured without them. Consider the following examples:

- `find_all_textures`: The determiner `all` signals a universal scope, clarifying that this identifier refers to the entire set of textures, not a subset.
- `on_start`: The preposition `on` reflects event-driven logic, indicating that the associated behavior is triggered at the start of execution.
- `warn_if_error`: The conjunction `if` embeds a conditional relationship, revealing that the action is contingent on an error occurring.

In each case, the closed-category term is essential to understanding the behavioral semantics of the identifier. Without these terms, the names are more ambiguous or less informative. While uncommon in aggregate, closed-category terms often signal precise intent and encode logical structure in compact forms.

Despite their potential significance, these terms have received almost no attention in prior software development naming research, which has focused primarily on open-category words (e.g., nouns, verbs). As a result, we lack foundational knowledge about when and how closed-category terms are used in code and what they contribute to program comprehension.

Understanding these naming patterns has clear implications: it can inform naming tools, guide educational resources, improve automated name generation, and help researchers characterize naming conventions more precisely. Closed-category terms may be uncommon, but we argue, in this paper, that their usage is not accidental; they significantly contribute to the meaning of identifier names, making it important to study them. More examples of identifiers containing closed-category terms can be found in Table 1.

3 Definitions & Grammar Pattern Generation

In this work, we analyze identifier names through the lens of **grammar patterns**, which are sequences of part-of-speech (PoS) tags assigned to the terms within an identifier. For example, the identifier `GetUserToken` is split into the terms `Get`, `User`, and `Token`, which are tagged as `Verb Noun-adjunct Noun`. This sequence, `V`

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

Abbreviation	Expanded Form	Examples
N	noun	stack, function, language
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	strings, identifiers, classes
NM (bold)	noun modifier	**employee**Name, **token**Parser
V	verb	run, execute, implement, develop
VM	verb modifier (adverb)	quickly, safely, eventually
PR	pronoun	she, he, her, him, it, we, us, they, them, I, me, you
D	numeral	1, 2, 10, 4.12, 0xAF
PRE	preamble*	Gimp, GLEW, GL, G

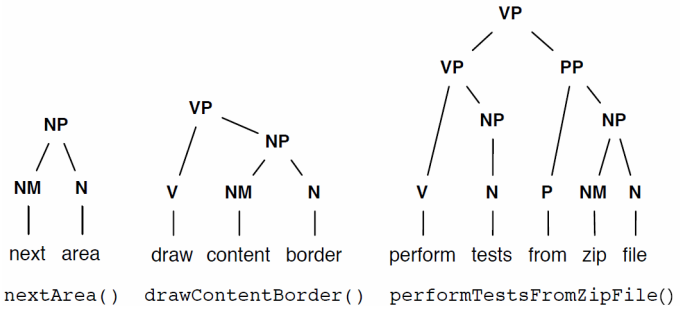

Fig. 1: Examples of noun, verb, and prepositional phrases

`NM N`, represents the identifier's grammar pattern. Crucially, this pattern generalizes across many identifiers: `RunUserQuery` and `WriteAccessToken` share the same structure, despite using different terms. Grammar patterns thus allow us to relate identifiers by their syntactic form.

We focus specifically on **closed-category grammar patterns**, which are patterns that contain at least one closed-class part of speech: a **preposition**, **determiner**, **conjunction**, or **numeral**. These categories are finite and rarely accept new terms, in contrast to open-class categories like nouns and verbs, which grow over time as new domains introduce new concepts. Despite their rarity in code, closed-category terms often signal behavioral relationships such as event triggers, quantification, or conditional logic, making them critical to study.

Part-of-Speech Tags. Table 2 lists the PoS tags used in this study. Most are drawn from standard linguistic categories. We highlight one custom tag that is central to our analysis:

– **Noun Modifier (NM):** Includes adjectives as well as *noun-adjuncts*—nouns used to modify another noun (e.g., `user` in `userToken`, or `content` in `contentBorder`).

Although standard PoS taggers do not typically distinguish noun-adjuncts, prior work shows their critical role in naming semantics [55].
- **Preamble (PRE):** A prefix used to convey structural or language-specific metadata, rather than domain semantics. Common examples include Hungarian-style markers such as `m_` for member variables, or project-level namespaces like `gimp` in `gimp_temp_file`; a practice especially common in C. For a complete typology and discussion, see [55]; we include preambles here since we do use them in the data set, but they are not the focus of this paper.

3.1 Phrasal Structures and Interpretation

While our analysis is based on PoS sequences rather than full parse trees, we draw on linguistic phrase structure to interpret identifier patterns. Specifically, we reference three example concepts to help the reader understand what we mean when we use the term 'phrase' with respect to grammar patterns:

- **Noun Phrase (NP):** A noun optionally preceded by one or more modifiers (e.g., `accessLog`, `userToken`, `windowTitle`).
- **Verb Phrase (VP):** A verb followed by a noun phrase, often representing an action on a specific entity (e.g., `getUserToken`, `drawContentBorder`).
- **Prepositional Phrase (PP):** A preposition followed by a noun phrase (e.g., `onClick`, `fromCache`).

These phrase structures help illustrate how grammar patterns support analysis of phrases. For instance, in `drawContentBorder`, the noun-modifier `content` refines the meaning of the head noun `border`, while the verb `draw` anchors the identifier as a behavior applied to that concept (i.e., draw applied to a specific type of border; a content-border). When closed-category terms appear, they may indicate when an action should occur (`onStart`), under what condition (`ifError`), or which entities are included (`allTextures`). Figure 1 shows examples of NP, VP, and VP-with-PP constructions as derived from grammar patterns.

4 Methodology

For our study, identifiers are collected from and analyzed in the following contexts: 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 local (to a function) or global variable. We use this terminology because it is consistent with srcML's terminology [20] for these variables, and we used srcML to collect identifiers. Therefore, to study closed-category grammar patterns, we group identifiers based on these five categories. The purpose of this categorization is to examine the closed-category grammar patterns based on their high-level semantic roles (e.g., class names have a different role than function names). We collected these identifiers from 30 open-source systems, which are listed in Table 3. These systems belonged to a curated dataset of engineered software projects, synthesized by Reaper [52], a tool that measures how well different projects adhere to software engineering practices, such as documentation and continuous integration.

Table 3: List of 30 open source systems included in study

Repo Link	Name	Primary Language	Data of most recent commit	C LOC	C++ LOC	Java LOC	Total LOC
https://github.com/liuliu/ccv	ccv	C	2023-07-19	279186	1908	0	281094
https://github.com/ropensci/git2r	git2r	C	2023-05-01	99956	0	0	99956
https://github.com/Juniper/libxo	libxo	C	2023-02-08	9361	0	0	9361
https://github.com/mgba-emu/mgba	mgba	C	2023-07-18	302865	25309	0	328174
https://github.com/naemon/naemon-core	naemon-core	C	2023-07-07	40991	0	0	40991
https://github.com/openvswitch/ovs	ovs	C	2023-07-19	268376	0	0	268376
https://github.com/igraph/rigraph	rigraph	C	2023-07-19	273973	32737	0	306710
https://github.com/toggl-open-source/toggldesktop	toggldesktop	C	2023-06-22	582087	269105	0	851192
https://github.com/irungentoo/toxcore	toxcore	C	2018-10-03	27443	0	0	27443
https://github.com/weechat/weechat	weechat	C	2023-07-20	197608	31175	0	228783
https://github.com/wireshark/wireshark	wireshark	C	2023-07-20	4171790	102848	0	4274638
https://github.com/BVLC/caffe	caffe	C++	2020-02-13	0	42856	0	42856
https://github.com/vgvassilev/cling	cling	C++	2023-07-18	57	28342	0	28399
https://github.com/ipkn/crow	crow	C++	2022-09-20	0	1434	0	1434
https://github.com/fakeNetflix/facebook-repo-ds2	ds2	C++	2019-07-17	367	27011	0	27378
https://github.com/freeminer/freeminer	freeminer	C++	2023-04-22	15090	130828	1077	146995
https://github.com/meta-toolkit/meta	meta	C++	2017-08-19	132	25451	0	25583
https://github.com/panda3d/panda3d	panda3d	C++	2023-06-13	44671	416212	175	461058
https://github.com/facebook/proxygen	proxygen	C++	2023-07-20	2676	70161	0	72837
https://github.com/s3fs-fuse/s3fs-fuse	s3fs-fuse	C++	2023-07-19	197	19582	0	19779
https://github.com/cglib/cglib	cglib	Java	2022-02-08	0	0	15187	15187
https://github.com/deeplearning4j/deeplearning4j	deeplearning4j	Java	2023-06-21	0	224997	696717	921714
https://github.com/apache/drill	drill	Java	2023-06-21	538	34591	626295	661424
https://github.com/google/guava	guava	Java	2023-07-18	0	0	356651	356651
https://github.com/immutables/immutables	immutables	Java	2023-06-16	0	0	69505	69505
https://github.com/dropwizard/metrics	metrics	Java	2023-07-20	0	0	31317	31317
https://github.com/igniterealtime/Openfire	Openfire	Java	2023-07-20	120	0	122186	122306
https://github.com/HubSpot/Singularity	Singularity	Java	2022-11-18	0	0	122183	122183
https://github.com/igniterealtime/Smack	Smack	Java	2023-04-26	0	0	125547	125547
https://github.com/igniterealtime/Spark	Spark	Java	2023-05-11	9	0	91886	91895
TOTAL				6317493	1484547	2258726	10060766

The set of systems has an average and median of 335,358 and 111,069 LOC, respectively. 11 of the systems are primarily C systems, 9 are mainly C++, and 10 are primarily Java. We chose systems that have tests and use continuous integration (CI) under the idea that these represent systems with at least some basic process for ensuring quality; Reaper is able to automatically determine which systems have both CI and tests. Our primary concern in selecting systems is that they represent different programming languages, follow basic quality procedures, and are large enough for us to collect a sufficient number of identifiers. Given this, our choice of systems is designed to ensure that the grammar patterns in this study are applied across at least the languages under study.

4.1 Detecting and Sampling Identifiers with Closed-Category Terms

Sampling identifiers that contain closed-category terms is challenging for two reasons: (1) They are relatively uncommon in production code, and (2) many such terms are ambiguous without context, making automatic tagging difficult. Table 4 shows the distribution of closed-category PoS tags present in a data set we constructed in prior work [55] versus the CCID, which was constructed explicitly to increase the population of closed-category PoS in the set. To address this, we implemented a two-phase sampling strategy: (1) filtering identifiers that *potentially* contain closed-category terms into candidate sets, and (2) manually verifying and annotating a statistically representative sample.

Phase 1: Filtering Candidate Identifiers

Table 4: Distribution of part-of-speech labels in Old Data Set and CCID

Old Data Set		CCID	
TAG	FREQUENCY	TAG	FREQUENCY
NM	1604 (45.2%)	N	1141 (31.58%)
N	1141 (32.1%)	NM	643 (17.79%)
V	305 (8.6%)	P	398 (11.02%)
NPL	238 (6.7%)	V	363 (10.04%)
PRE	105 (3%)	DT	308 (8.52%)
P	94 (2.6%)	D	283 (7.83%)
D	27 (0.8%)	PRE	217 (6.00%)
DT	15 (0.4%)	NPL	142 (3.93%)
VM	13 (0.4%)	VM	69 (1.91%)
CJ	8 (0.2%)	CJ	50 (1.38%)
Total	3550	Total	3614

Table 5: Distribution of Tags in Candidate and Verified (Manually-annotated) data set

	CJ		**DT**	
	Candidate	Verified	Candidate	Verified
Attribute	66 (28.09%)	6 (12.24%)	78 (24.92%)	84 (27.54%)
Declaration	62 (26.38%)	10 (20.41%)	79 (25.24%)	85 (27.87%)
Parameter	44 (18.72%)	6 (12.24%)	78 (24.92%)	58 (19.02%)
Function	63 (26.81%)	27 (55.10%)	78 (24.92%)	78 (25.57%)
Class	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
Total	235	49	313	305
	D		**P**	
	Candidate	Verified	Candidate	Verified
Attribute	80 (21.98%)	62 (23.40%)	89 (24.45%)	103 (26.96%)
Declaration	81 (22.25%)	70 (26.42%)	88 (24.18%)	73 (19.11%)
Parameter	81 (22.25%)	77 (29.06%)	89 (24.45%)	60 (15.71%)
Function	80 (21.98%)	41 (15.47%)	88 (24.18%)	140 (36.65%)
Class	42 (11.54%)	15 (5.66%)	10 (2.75%)	6 (1.57%)
Total	364	265	364	382

We began with the CCID corpus, which contains 279,000 unique identifiers from production code. To collect these 279K identifiers, we used the srcML identifier getter tool [1] on the srcML archives resulting from running srcML [20] on the system repository directories (Table 3). To identify candidate sets:

[1] https://github.com/SCANL/srcml_identifier_getter_tool

Table 6: Balanced population of identifiers per context

Context	Sample Population
Attribute	312 (24.47%)
Declaration	306 (24.00%)
Function	313 (24.55%)
Class	52 (4.08%)
Parameter	292 (22.90%)
Total	1275

- **Numerals (D):** We selected identifiers containing at least one digit, using Python's `isdigit()` functionality. Numerals are easier to detect automatically and are unambiguous in token form. However, there are cases where a numeral will be annotated as part of another category. For example, *str2int* uses the numeral 2 as a preposition (to).
- **Determiners (DT), Conjunctions (CJ), and Prepositions (P):** We constructed lexicons for each category using curated lists of common English terms[2][3][4]. We then filtered for identifiers containing component words (i.e., split tokens) that matched a word in one of these lists. This approach is viable only because these categories are closed and finite in vocabulary.

This filtering process produced the following candidate counts:

- 602 identifiers candidate conjunctions
- 1,693 identifiers candidate determiners
- 3,383 identifiers candidate prepositions
- 4,630 identifiers candidate numerals

We use the term *candidate* because these filters do not account for context or usage, and thus include false positives. Still, they serve as an upper bound for the prevalence of each category in the corpus. Based on this, we estimate the proportion of identifiers containing each term type as follows:

- **0.2%** (602/279,000) contain conjunctions
- **0.6%** (1,693/279,000) contain determiners
- **1.2%** (3,383/279,000) contain prepositions
- **1.7%** (4,630/279,000) contain numerals

Phase 2: Balanced Sampling and Annotation

Using a 95% confidence level and a 5% margin of error, we computed minimum sample sizes for each category. For example, a 95 and 5 sample for conjunctions (602 identifiers) is 235:

- **CJ:** 235 candidate conjunction identifiers
- **DT:** 313 candidate determiner identifiers
- **P:** 345 candidate preposition identifiers
- **D:** 355 candidate numeral identifiers

[2] `https://en.wikipedia.org/wiki/List_of_English_determiners`
[3] `https://www.englishclub.com/grammar/prepositions-list.php`
[4] `https://7esl.com/conjunctions-list/`

Before manual annotation, we stratified the candidate identifiers by their program context:

- Function names
- Parameters
- Attributes (i.e., class members)
- Function-local declarations
- Class names

In some contexts, such as parameters and especially class names, terms were underrepresented due to the natural scarcity of closed-category terms in those positions. To increase representation from underrepresented contexts, we attempted to oversample relevant subgroups. However, even with oversampling, the absolute number of qualifying identifiers (e.g., a parameter containing a conjunction) remained low. As our sampling was driven by the presence/population of closed-category terms in general, rather than population within our program contexts under study, we opted not to artificially balance the dataset further.

After sampling and stratification, we obtained a total of 1,275 identifiers across the four categories in our **candidate set**:

- **364** candidate preposition identifiers
- **363** candidate numeral identifiers
- **313** candidate determiner identifiers
- **235** candidate conjunction identifiers

Following manual verification and annotation (described in Section 4.2), we retained only those identifiers that were confirmed to contain closed-category terms. Table 5 summarizes both the sampled totals and the verified counts. The final dataset consists of 1,001 identifiers confirmed to include at least one closed-category term. **These identifiers comprise the CCID**. Table 6 shows the CCID, but broken down by program context instead of closed-category type.

Annotation Notes. Some tools exist for part-of-speech tagging of source code identifiers (e.g., the ensemble tagger from [57]), but these are slow at scale and are trained on datasets that underrepresent closed-category terms. For example, in our prior dataset [55], used to train the aforementioned tagging approach [57]; conjunctions, determiners, and prepositions made up only 0.2%, 0.4%, and 2.6% of tags, respectively. Thus, we found manual annotation necessary to ensure sufficient coverage and correctness for our study.

As we are primarily concerned with production code, and prior work shows that test name grammar patterns differ from production names [61], we did not collect any identifiers containing the word 'test', or that appeared in a clearly marked test file or directory. In addition, note that Table 4 counts tags at the *word* level (e.g., CJ CJ N counts two CJ tags), whereas Table 5 counts tags at the *identifier* level (e.g., one identifier with multiple CJ tags counts as one). This explains occasional mismatches between sampled and actual tag distributions.

4.2 Manual Process for Annotating Part-of-Speech

Initially, one author (annotator) is assigned annotate each identifier in the CCID with its grammar pattern. The annotator has experience annotating identifiers

Table 7: Most common Patterns

Determiner		Conjunction		Preposition		numeral	
DT N	109 (35.74%)	N CJ N	6 (12.24%)	P N	70 (18.32%)	N D	125 (47.17%)
DT NM N	40 (13.11%)	V CJ N	3 (6.12%)	P NM N	31 (8.12%)	NM N D	33 (12.45%)
DT NPL	20 (6.56%)	CJ NM	2 (4.08%)	P V	13 (3.40%)	N D N	9 (3.40%)
DT V	7 (2.30%)	NM CJ NM	2 (4.08%)	N P N	12 (3.14%)	PRE N D	8 (3.02%)
DT NM NM N	6 (1.97%)	V CJ V	2 (4.08%)	V P N	12 (3.14%)	V N D	8 (3.02%)
N DT	6 (1.97%)	V N CJ N	2 (4.08%)	P	10 (2.62%)	NPL D	6 (2.26%)
V DT	6 (1.97%)	CJ	1 (2.04%)	NM N P N	9 (2.36%)	N D NM N	3 (1.13%)
DT NM NPL	5 (1.64%)	CJ DT N	1 (2.04%)	V P	9 (2.36%)	V NM N D	3 (1.13%)
V DT N	5 (1.64%)	CJ NM N	1 (2.04%)	N P	8 (2.09%)	N D NPL	2 (0.75%)
DT	4 (1.31%)	CJ V	1 (2.04%)	NPL P N	8 (2.09%)	N P D NPL	2 (0.75%)

with part-of-speech (PoS) tags from prior work [55, 61]. The process is as follows: The annotator is given a split (using Spiral [42]) identifier, along with its type, file path, and line number, to facilitate easy identification of the identifier in the original code. The annotator is permitted to examine the source code from which the identifier originated, if necessary. The annotator is asked to additionally identify and correct mistakes made by Spiral. When the annotator is finished, two additional annotators are asked to validate (agree or disagree) with the annotations created by the original annotator. Any disagreements are discussed and fixed, if required. Furthermore, a fourth annotator assigned their own annotations, which are then compared to the original annotator's work. Again, disagreements are discussed and fixed. An example disagreement is with the identifier *where_len*, which is a tricky one because 'where' is typically an adverb or conjunction. However, in this case 'where' is a reference to a void pointer variable called 'where' within the code. So 'where_len' is the length of the memory this pointer points to, making 'where' a noun-adjunct in this case; describing the type of length. Thus, its grammar pattern is NM N. **The Fleiss' Kappa for this process was .916**.

We did not expand abbreviations for a couple reasons. The first is that some abbreviations are more meaningful than their expanded terms (e.g., HTTP, IPv4, SSL) due to their frequent use in abbreviated form by the community. The second reason is that abbreviation expansion techniques are not widely available and vary widely in terms of effectiveness on different types of terms [56, 75]. Therefore, a realistic worst-case scenario for developers and researchers is that no abbreviation-expansion technique is available to use, and their PoS taggers must work in this worst-case scenario. Whenever we recognize one, we do not split domain-term abbreviations (e.g., Spiral will make IPv4 into IPv 4; we corrected this to IPv4). We do this because it is the view of the authors that they should be recognized and appropriately tagged in their abbreviated (i.e., their most common) form.

5 Evaluation of RQ1: What behavioral roles do closed-category terms play in source code identifiers?

Our evaluation aims to establish, through RQ1 and RQ2: 1) how closed-category terms are used to convey differing types of program behavior, 2) the typical grammatical structure of identifiers containing closed-category terms, and 3) how

closed-category term distributions differ across programming context, language, and system domains. This first research question investigates the semantic role of closed-category grammatical patterns in identifier naming. We focus on four closed-category part-of-speech types: prepositions, numerals, determiners, and conjunctions. We present our findings by (1) describing each category's semantic function using axial codes, (2) summarizing behavioral trends via selective coding, and (3) highlighting shared trends through cross-category synthesis.

5.1 Methodology: Manual Process for Behavioral Annotations

We employed an approach inspired by Straussian grounded theory to analyze variable names in source code and their relationship to program behavior. This multi-phase coding process involved four annotators, combining individual annotations with iterative validation and synthesis to construct a theory grounded in observed naming patterns. The sample used in this study is a subset of the CCID described in Section 4. To construct this subset, we took the top 10 most common grammar patterns (Table 7) and collected all identifiers that followed these patterns; randomly selecting the 10th grammar pattern if multiple patterns had the same frequency. These represent the most common names (i.e., from the perspective of grammatical structure) used in the data set. **This totals to 618 identifiers.**

Four annotators participated in the process, comprising both faculty and graduate students with prior experience in natural language processing and software engineering research. All annotators had previously worked on part-of-speech annotation tasks. Before formal annotation began, the team conducted a one-hour training and calibration session to discuss the guidelines, walk through examples, and establish expectations and deadlines.

Coding Platform. Annotations are conducted collaboratively using a shared Google Sheets document. Each row in the sheet contained an identifier along with contextual metadata, including:

- **Identifier Name**
- **Source Code Context**
- **Programming Language**
- **GitHub Commit Link**
- **Split Identifier Name** (tokenized form)
- **Grammar Pattern** (POS sequence)
- **Notes** (for open coding and memoing)
- **Axial Code** (for grouping behavioral patterns)

Open coding and memoing are captured directly in the `Notes` column. The final axial codes were recorded in the corresponding column once annotators had synthesized their observations.

Phase 1: Familiarization. All annotators reviewed the dataset to build familiarity with the variable names, associated grammar patterns, and program contexts. They discussed ambiguous or novel constructions in group chats to align interpretations and maintain consistency.

Phase 2: Open Coding. Annotators examined each variable name in its context and assigned a free-form behavioral interpretation based on how the variable is used in the surrounding code. These open codes and rationale are documented in the `Notes` column. The goal was to capture a grounded understanding of what each identifier conveyed, informed by both linguistic structure and program behavior.

Phase 3: Axial Coding. Annotators grouped similar open codes into higher-level axial codes, focusing on patterns where particular grammatical structures consistently aligned with specific behavioral roles. These axial codes captured mid-level abstractions (e.g., *State Variables*, *Event Triggers*), and were documented in the spreadsheet alongside notes justifying the grouping where needed. Each annotator's axial codes were reviewed by a different annotator for validation. This cross-review process involved reading both the open codes and the proposed axial codes, discussing disagreements, and refining the categories until consensus was reached. The Fleiss' Kappa for this phase was: **.971 for numerals**, **.996 for Determiners**, **.976 for Prepositions**, and **1.0 for Conjunctions**.

Phase 4: Selective Coding. One annotator synthesized the final, validated axial codes across all annotations and constructed a set of selective codes representing core theoretical categories that linked grammar structure to program intent. These selective codes were then shared with the remaining annotators, who were asked to evaluate whether they reflected the themes and relationships they had observed during their own coding work. Annotators agreed or suggested revisions to finalize the theory.

5.2 Numerals in Identifiers

Overview. Numerals in identifiers act as compact, semantic indicators of structure, ordering, or version. They are also often used to disambiguate entities and encode numeric conventions. Their meaning is typically inferred through domain knowledge, making them powerful, but potentially hard to understand for those without the requisite domain knowledge.

Axial Codes. We created a dual-axis framework for interpreting the meaning of numerals, inspired by a single-axis framework we created in prior work on numerals in identifiers [65]. This framework reflects our observation that numerals contribute information in two distinct ways: (1) the **role** they play within the local context (e.g., indexing, versioning), and (2) the **source of meaning** they draw from, which is often external to the immediate source code scope (e.g., domain conventions, technical standards). **Every numeral in the set has both a role and a source of meaning; they must be combined to fully understand the numeral.** We put an 'x' between each combination of 'Role' and 'Source of Meaning' Axial Code.

- **Role:** What functional purpose the numeral serves in the identifier.
  - **Distinguisher:** The numeral differentiates conceptually similar entities, typically to avoid name collision errors from the compiler (e.g., `arg1`, `tile2`).

  - **Version Identifier:** The numeral encodes versioning information such as protocol revisions or data format versions (e.g., `http2`, `v1`).
- **Source of Meaning:** Where the interpretation of the numeral originates, typically via convention, tooling, or domain-specific logic.
  - **Auto-Generated:** The numeral is added automatically by tools, compilers, or naming systems to avoid conflicts (e.g., `var1_2`, `jButton3`).
  - **Human-Named Convention:** The numeral's meaning is primarily derived from ad hoc developer intent and is not more complex than distinguishing entities manually (e.g., `str1`, `feature2`).
  - **Locally Specific Concept:** The numeral conveys project- or context-specific information, often related to coordinate systems, data structures, or memory layouts (e.g., `m33` for matrix row 3 col 3).
  - **Technology Term / Standard:** The numeral is part of a recognized domain-specific label, format, or protocol (e.g., `HTTP2`, `Neo4j`).

Role x Source of Meaning

1. **Distinguisher × Human-Named Convention (122 items)**
   Description: This group captures identifiers that use manually assigned numeric suffixes to distinguish conceptually and lexically similar entities.
   Examples: `host1` (first of 2 host variables), `e8` (element 8 in a parameter list)
   Grammar patterns:
   - N D (73)
   - NM N D (21)
   - V N D (6)
   - NPL D (6)
   - N D N (5)
   - PRE N D (3)
   - N D NM N (2)
   - P D (2)
   - PRE NM N D (2)
   - V NM N D (2)
2. **Distinguisher × Locally Specific Concept (45 items)**
   Description: This group captures identifiers where numerals encode positional or logical roles based on system-specific conventions, such as grid layout or data structure indexing.
   Examples: `dist2` (squared distance calculation), `col1` (first column of a matrix)
   Grammar patterns:
   - N D (32)
   - NM N D (5)
   - PRE N D (3)
   - N D N (2)
   - NM N D P D (2)
   - V N D (1)
3. **Distinguisher × Technology Term / Standard (17 items)**
   Description: This group captures identifiers that include numerals as part of standardized or domain-specific naming conventions, often encoding formats or specifications.
   Examples: `b1110` (binary for UTF8 byte sequences), `count32` (32-bit count

value)
Grammar patterns:
- N D (7)
- NM N D (5)
- PRE N D (2)
- N D N (1)
- V N D (1)
- V NM N D (1)

4. **Version Identifier × Technology Term / Standard (9 items)**
Description: This group captures identifiers where the numeral signals the version number of a protocol, tool, or technology component.
Examples: `gw6` (gateway addr for IPV6), `httperf2` (version 2 of the `httperf` tool)
Grammar patterns:
- N D (5)
- NM N D (2)
- N D N (1)
- N D NM N (1)

5. **Distinguisher × Auto-Generated (8 items)**
Description: This group captures identifiers that are automatically suffixed with a numeral to ensure uniqueness, often generated by tools or compilers.
Examples: `field37`, `field4` (numbers are generated to avoid name collissions)
Grammar patterns:
- N D (8)

Example. Consider the identifier `m34`, which appears in the context of a matrix operation. To fully interpret the numeral `3` in this name, we must consider both its Role and its Source of Meaning. Semantically, the numeral serves as a *Distinguisher*; uniquely identifying this variable apart from its siblings (such as m32 and m31). However, its complete interpretation depends on its *Locally Specific Concept* source: the developers have an internal convention that `3` refers to the row index, while `4` refers to the column. Without knowing the Source of Meaning, the numbers can only be interpreted as distinguishing one identifier from another; the meaning of the numerals would remain ambiguous. This illustrates how both axes work together—Role tells us *what the numeral is doing*, while Source of Meaning tells us *how to interpret the value*.

Selective Coding Insight. Numerals serve as semantic compression tools in source code: conveying versioning, layout, ordering, or configuration state using a minimal footprint. Their power lies in the idea that both the identifier's author and readers share a certain level of domain knowledge, and thus can understand the meaning of the numeral. Whether distinguishing hosts (`host1`, `host2`), signaling protocol versions (`http2`), or denoting matrix dimensions (`m33`), numerals rely on prior knowledge to be effective, this makes them:

- **Easy to understand** when used in well-known conventions (e.g., `3D`, `utf8`)
- **Hard to understand** when overused without documentation or when the reader lacks background information/experience that the author assumed they would have

Numerals are *structural shortcuts* in the mental models of developers; a quick way to convey a lot of information in a small number of characters.

5.3 Prepositions in Identifiers

Overview. Prepositions in identifiers express spatial, temporal, or logical relationships. They are the most versatile (i.e., most axial codes) and frequently used closed-class grammatical structure in our dataset. Prepositions typically convey transformation, control conditions, event triggers, source origin, or context membership. Because only a subset of these are dual-axis (Boolean Flow), we inline the definitions with our examples, unlike with numerals, where we separate them.

Axial Codes. Through axial coding, we identified several recurring behavioral roles that prepositions play in identifier names. These axial codes describe the functional semantics conveyed by the preposition within the naming context:

1. **Type Casting / Interpretation (38 items)**
   Definition: This group captures identifiers that signify transformation from one type, format, or abstraction to another.
   Examples: `str_2_int`, `as_field`
   Grammar patterns:
   - P N (18)
   - P NM N (9)
   - N P N (4)
   - V P N (2)
   - P NM NM N (2)
   - P V (1)
   - NM N P N (1)
   - V P (1)
2. **Position / Ordering in Time or Space (28 items)**
   Definition: This group captures identifiers that indicate relative position or sequencing within a spatial, temporal, or execution context.
   Examples: `before_major`, `after_first_batch`
   Grammar patterns:
   - P N (8)
   - P (4)
   - N P N (3)
   - P NM N (3)
   - V P N (3)
   - P V (2)
   - V P (2)
   - NM N P N (2)
   - N P (1)
3. **Boolean Flow / Control Flag (26 items)**
   Definition: This group captures identifiers that encode boolean flags which both guard execution and describe the behavior they enable. This group is somewhat special, as their name implies other axial codes, but they are boolean variables. Thus, many of the identifiers in this group are dual-axis, where the

1st axis is boolean, and the 2nd is one of the other preposition axes. These variables are typically guards, used in branching logic that:
  - Activate based on *position or sequencing* (e.g., `after_equals`)
  - Govern *strategy or type casting/interpretation behavior* (e.g., `for_backprop`, `as_array`)
  - Reflect *data provenance or deferred logic* (e.g., `from_docker_config`, `wait_for_reload`)

  Examples: `obsess_over_host`, `for_backprop`
  Grammar patterns:
  - P N (12)
  - P NM N (6)
  - V P N (2)
  - V P (2)
  - N P N (2)
  - N P (1)
  - P (1)

4. **Data Source / Origin (20 items)**
  Definition: This group captures identifiers that refer to the source from which data or configuration is retrieved.
  Examples: `from_context`, `from_id`
  Grammar patterns:
  - P N (10)
  - P NM N (1)
  - N P N (2)
  - P (3)
  - NM N P N (1)
  - V P (1)
  - N P (2)

5. **Event Callback / Trigger (17 items)**
  Definition: This group captures identifiers that define behavior executed in response to user or system events.
  Examples: `on_reason`, `on_start`
  Grammar patterns:
  - P N (6)
  - P NM N (5)
  - P NM NM N (4)
  - V P N (1)
  - NM N P N (1)

6. **Deferred Processing / Pending Action (13 items)**
  Definition: This group captures identifiers that signal actions or data awaiting future handling.
  Examples: `to_ack`, `to_count`
  Grammar patterns:
  - P V (10)
  - P N (2)
  - P NM N (1)

7. **Unit-Based Decomposition / Measurement (11 items)**
  Definition: This group captures identifiers that describe per-unit measurement, processing, or aggregation.

Examples: `down_time`, `size_in_datum`
Grammar patterns:
- NPL P N (8)
- P N (1)
- N P N (1)
- NM N P N (1)

8. **Purpose / Role Annotation (10 items)**
Definition: This group captures identifiers that clarify the functional role or use-case of a value.
Examples: `for_avg`, `for_class`
Grammar patterns:
- P N (6)
- NM N P N (2)
- P NM N (1)
- V P (1)

9. **Data Movement / Transfer (9 items)**
Definition: This group captures identifiers that represent movement of data or control between locations, buffers, or components.
Examples: `to_repo`, `to_header`
Grammar patterns:
- P N (3)
- N P (3)
- P NM N (1)
- NM N P N (1)
- P NM NM N (1)

10. **Operation Basis / Strategy (8 items)**
Definition: This group captures identifiers that describe the method, or trait that determines how operations may/should be carried out.
Examples: `extend_by_hexahedron`, `with_unary_operator`
Grammar patterns:
- P N (2)
- P NM N (2)
- V P N (2)
- P (1)
- V P (1)

11. **Membership / Peer Grouping (7 items)**
Definition: This group captures identifiers that signal inclusion in a group, scope, or set of peer entities.
Examples: `in_neighbour_heap`, `in_for`
Grammar patterns:
- P (2)
- P N (1)
- P NM N (1)
- V P N (1)
- V P (1)
- N P (1)

12. **Mathematical / Constraint Context (2 items)**
Definition: This group captures identifiers that encode numerical limits, bounds, or ratios that constrain behavior.

Examples: `over_size`, `vmax_over_base`
Grammar patterns:
– P N (1)
– N P N (1)

Selective Coding Insight. Prepositions in identifier names serve as compact, highly expressive relational markers. Across the dataset, prepositions consistently support four core semantic roles:

– **Transformation and Directionality**: Prepositions like `to`, `from`, and `as` signal type casting, movement, or format conversion.
– **Execution and Conditional Control**: Prepositions such as `after`, `on`, and `for` often signal when or whether an action should occur, especially within event-driven operations and boolean flags that gate execution.
– **Role and Configuration Semantics**: Prepositions like `with`, `by`, and `in` clarify how values contribute to a process or how behavior is scoped or grouped.
– **Quantification and Unit-Based Aggregation**: Prepositions such as `per` and `in` describe how quantities are measured, normalized, or decomposed across units (e.g., `iterations_per_sample`, `size_in_datum`).
– **Future-Intent or Deferred Action**: Especially with `to`, some identifiers encode pending or scheduled behavior (e.g., `to_merge`, `wait_for_reload`).

Importantly, boolean flags that include prepositions do not form a distinct behavioral class, but instead overlay these four functions; gating type conversions, controlling source-based logic, or scoping strategies. These flags act as behavioral summaries, where the identifier directly reflects the guarded behavior (e.g., `send_to_buffer` reflects that the guarded code sends data to a buffer).

In short, prepositions make invisible system relationships visible. They map the logic of control, transformation, and association directly into identifier structure, enabling expressive, intention-revealing naming in complex systems.

5.4 Determiners in Identifiers

Overview. Determiners in identifiers help interpret values in relation to a set. They often signal positional reasoning, filtering criteria, relative thresholds, control flow, or scoping rules. In our analysis, we treat terms like `next` and `last` as **determiners**, even though they are typically categorized as adjectives in general English. In source code, however, these terms function more like determiners because **they specify a particular entity within a sequence or collection rather than merely describing its properties**. For example, the `next` pointer in a linked list does not describe a type of pointer, but rather identifies the specific node that follows in the structure. In this way, such terms serve a determinative function.

Axial Codes. We identified the following eight categories of determiner-based behavior:

1. **Temporal / Most Recent Element (60 items)**
Definition: This group captures identifiers that refer to the most recently computed, stored, or observed value, often used for computing prior state, and in

sequence-based data structures.
Examples: `last_bucket`, `last_builder`
Grammar patterns:
- DT N (32)
- DT NM N (19)
- DT NM NM N (4)
- DT V (2)
- V DT N (2)
- DT NPL (1)

2. **Temporal / Upcoming Element (54 items)**
Definition: This group captures identifiers that denote the next item in a sequence or timeline, often used in look-ahead and sequence-based data structures.
Examples: `next_tex`, `next_bar`
Grammar patterns:
- DT N (35)
- DT NM N (9)
- DT V (3)
- N DT (3)
- DT NPL (2)
- DT NM NM N (1)
- V DT N (1)

3. **Population / Subpopulation Reference (42 items)**
Definition: This group encompasses identifiers that reference a population or subset, typically using quantifiers such as `all`, `any`, or `some` to guide iteration, filtering, or policy logic.
Examples: `any_diffuse`, `all_set`
Grammar patterns:
- DT NPL (13)
- DT N (9)
- V DT (6)
- DT NM NPL (4)
- V DT NPL (4)
- DT NM N (2)
- N DT (2)
- DT V (1)
- V DT N (1)

4. **Immediate Context Reference (26 items)**
Definition: This group captures identifiers that refer to the current instance, scope, or runtime context—emphasizing locality, such as `this`, `another`, or `a`.
Examples: `this_node`, `another_id`
Grammar patterns:
- DT N (17)
- DT NM N (6)
- DT NM NM N (1)
- N DT (1)
- V DT N (1)

5. **Negation / Exclusion Flag (18 items)**
Definition: This group captures identifiers that indicate something is explicitly

disabled, excluded, or absent; commonly using `no` to toggle features or signal null conditions.
Examples: `no_callback`, `no_log`
Grammar patterns:
- DT N (12)
- DT NM N (2)
- DT NPL (2)
- DT NM NPL (1)
- DT V (1)

6. **Quantity Threshold / Optional Extensibility (4 items)**
Definition: This group captures identifiers that express minimum thresholds, or the possibility of extending beyond a baseline.
Examples: `enough_memory`, `more_data`
Grammar patterns:
- DT N (2)
- DT NPL (2)

7. **Default / Fallback Value Representation (2 items)**
Definition: This group captures identifiers that represent placeholder or fallback values, used when a field must be filled or a default condition must be satisfied.
Examples: `a_void`, `no_val`
Grammar patterns:
- DT N (2)

8. **Boolean Multi-Condition Test (2 items)**
Definition: This group captures boolean identifiers representing conjunctions of multiple conditions, usually requiring all to be satisfied (e.g., both X and Y must be true).
Examples: `both_empty_selection`, `both_NonEmpty_Selection`
Grammar patterns:
- DT NM N (2)

Selective Coding Insight. Determiner-based identifiers help interpret values in relation to a set—by signaling *position*, *filtering criteria*, *thresholds*, or *scoping rules*. These are closed-category terms that enable programmers to express *set logic*, *entity selection*, and *relative capacity or validity*. They typically support:

- **Positional reasoning** (`next`, `last`, `this`): Indicates where a value occurs in a temporal or structural sequence, helping to track state progression, history, or future execution.
- **Population membership and filtering** (`some`, `any`, `each`, `least`, `which`): Refers to selecting or referencing members within a larger set, expressing scope, quantification, or comparison.
- **Thresholding and extensibility** (`enough`, `more`, `additional`): Indicates whether a minimum condition is met or whether more values can be included beyond a base requirement.
- **Identity negation or fallback** (`no`, `none`, `a`, `without`): Flags exclusion, absence, or placeholder values—often tied to feature toggles or default logic.

5.5 Conjunctions in Identifiers

Overview. Conjunction-based identifiers are rare but expressive. They signal compound behavior, dual-mode interfaces, or gated logic—often making hidden control flow or semantic relationships visible. Their rarity likely stems from the fact that developers often express conjunctions in logic rather than names. But when used, they highlight either an intent to emphasize control-flow behavior or to capture structural duality within a single name.

Axial Codes. We identified seven categories of conjunctional behavior, each reflecting a different type of pairing, conditionality, or combination:

1. **Data Pair / Composite Value (7 items)**
   Definition: This group captures identifiers that hold or refer to two values used together or in alternation, typically for a shared behavioral role or composite purpose.
   Examples: `data_or_diff`, `function_and_data`
   Grammar patterns:
   – N CJ N (6)
   – V CJ N (1)
2. **Guarded Action / Conditional Enablement (6 items)**
   Definition: This group captures identifiers that encode actions gated by internal logic; executing only if a condition is satisfied. The conjunction expresses conditional enablement or guarded behavior.
   Examples: `if_present`, `if_unique`
   Grammar patterns:
   – CJ NM (2)
   – V CJ N (2)
   – V CJ V (1)
   – V CJ VM P (1)
3. **Combined Action / Sequential Behavior (3 items)**
   Definition: This group captures identifiers that describe a sequence of operations performed together, often representing merged behaviors.
   Examples: `hash_and_save`, `print_and_free_json`
   Grammar patterns:
   – V CJ V (1)
   – V CJ V N (1)
   – V N CJ N (1)
4. **Shared Interface for Alternatives (1 item)**
   Definition: This group captures identifiers that define a shared interface or behavior over mutually exclusive alternatives, with the conjunction indicating a choice, not a combination.
   Example: `generate_key_or_iv`
   Grammar pattern:
   – V N CJ N (1)
5. **Combined Configuration / UI Concept (1 item)**
   Definition: This group captures identifiers that refer to compound interface or configuration concepts, often blending multiple traits into a unified design or behavioral setting.

Example: `look_and_feel`
Grammar pattern:
– NM CJ NM (1)

6. **Boolean Concept Name (1 item)**
Definition: This group captures identifiers that encode a named logical or boolean relationship, usually by treating the conjunction itself as a symbolic concept.
Example: `and`
Grammar pattern:
– CJ (1)
7. **Boolean Multi-Condition Test (1 item)**
Definition: This group captures identifiers that evaluate multiple conditions simultaneously; typically for readiness or validation checks, returning true only if all constraints are met.
Example: `null_or_empty`
Grammar pattern:
– NM CJ NM (1)

Selective Coding Insight. Conjunction-based identifiers are especially useful when modeling:

– **Duality**: Representing more than one entity or mode simultaneously (e.g., `input_and_output`, `key_or_iv`)
– **Mutual Exclusion**: Encoding choices between alternatives—only one active at a time (e.g., `stream_or_cache`)
– **Preconditions**: Embedding logic into the name that would otherwise be hidden in branching statements (e.g., `load_if_needed`, `trigger_if_active`)

Conjunctions are the rarest category in our data, and while it is difficult to draw firm conclusions about them, it is clear that 'and', 'or', and 'if' are go-to conjunctions, particularly for Data Pairs and Guarded actions.

5.6 Cross-Category Synthesis

Across numerals, determiners, prepositions, and conjunctions, developers use closed-class grammatical structures to encode compact, behavior-rich semantics in identifiers. While each PoS category exhibits distinct tendencies, analysis of grammar patterns reveals broader functional themes and stylistic consistencies across categories.

Boolean Semantics and Execution Control. Our first cross-category behavior is the use of closed-class elements to encode boolean conditions, execution control, or logical gating:

1. Determiners such as *no*, *some*, *this*, and *both* signal presence, exclusion, or multi-condition boolean evaluation.
2. When used as booleans, Prepositions like *as*, and *with* tend to guard sections of code that implement the behavior described in the identifier name.

3. Conjunctions surface explicitly in guarded or compound logic names (e.g., load_if_enabled, both_ready) using patterns like V CJ N, NM CJ NM.

Interestingly, booleans appear in all three of these contexts, but each is a different flavor; a way of expressing behavior that is unique to the closed-category terms used in the boolean identifier.

Control Flow and Event Signaling Across Categories. Closed-category terms across all four categories reflect a tendency to encode temporal, reactive, or preconditioned behavior:

1. Prepositions like on, before, after, and by appear in structures such as P N and V P N, signaling timing, triggers, or basis for operation.
2. Conjunctions explicitly model control conditions (if, and) or mutual exclusivity (or), often appearing in V CJ N or N CJ N structures.
3. Determiners frequently encode sequence through next and last, realized in DT N and DT NM N patterns.
4. Numerals imply procedural differentiation (method1, step2) or timeline indexing when appearing in coordinated identifiers (m31, m32).

These names act as micro-control structures, embedding state transitions and flow logic directly into identifier names to help the reader understand when or how an identifier will/should be used.

Multi-Dimensional Semantic Layering. Grammar pattern analysis highlights how identifiers stack multiple behavioral dimensions:

1. Prepositions convey direction, transformation, measurement, and order
2. Determiners convey selection, quantity, and scope
3. Numerals embed indexing, uniqueness, and domain roles
4. Conjunctions encode logic composition and structural alternatives.

These layered forms serve as semantic shortcuts to convey complex behavior with minimal words. They compress conditions, transformations, order, and relationships into concise forms that aim to assist program comprehension and understanding without excess verbiage.

Finally, the **grammar patterns** observed across our axial codings provide structural insight into how behavioral semantics are composed. When the closed-category term appears as the first token in a grammar pattern, such as in `DT NM N` or `P NM N`, it typically modifies or qualifies a single operand, forming a unary relation (e.g., temporal status or transformation of a noun phrase). In contrast, when the closed term is flanked by open-class terms, such as in `N P N` or `N CJ N`, the structure reflects a binary relation: two operands connected through a behavioral or logical relationship (e.g., data flow or choice).

By combining our axial and selective codes with these syntactic patterns, we gain a fuller picture of identifier meaning: the open-category terms indicate *which* entities are involved, while the closed-category term signals *how* they are related or behave with respect to one another.

Table 8: Top 10 terms per closed-category part-of-speech tag

Determiner	Conjunction	Numeral	Preposition
last (79, 25.65%)	and (18, 36.00%)	1 (77, 27.21%)	to (76, 19.10%)
next (69, 22.40%)	if (16, 32.00%)	2 (71, 25.09%)	for (37, 9.30%)
all (52, 16.88%)	or (13, 26.00%)	0 (20, 7.07%)	on (36, 9.05%)
no (31, 10.06%)	than (1, 2.00%)	3 (17, 6.01%)	as (35, 8.79%)
this (21, 6.82%)	since (1, 2.00%)	4 (15, 5.30%)	from (33, 8.29%)
each (7, 2.27%)	when (1, 2.00%)	16 (13, 4.59%)	in (25, 6.28%)
the (6, 1.95%)	-	8 (8, 2.83%)	after (21, 5.28%)
more (5, 1.62%)	-	6 (6, 2.12%)	2 (to) (17, 4.27%)
a (4, 1.30%)	-	64 (4, 1.41%)	of (15, 3.77%)
some (4, 1.30%)	-	32 (3, 1.06%)	with (14, 3.52%)

5.7 Summary of RQ1

Through qualitative analysis of closed-category terms in identifiers, we have uncovered and explored how these compact grammatical forms play a central role in expressing program behavior. Each part-of-speech category contributes distinct semantic functions, ranging from transformation and scoping to control flow and logical composition.

Together, they reveal how developers construct concise, behavior-rich identifiers that encode structure, timing, intent, and logic. Whether signaling preconditions (`load_if_enabled`), alternatives (`data_or_diff`), state (`last_bucket`), or structural roles (`col1`), these terms form a functional lexicon that bridges source code, cognition, and context.

6 Evaluation of RQ2: How do closed-category terms correlate with structural, programming language, and domain-specific contexts in software?

One interesting aspect of *closed-category terms* is that they appear in different contexts within source code with varying frequency. This variation provides insight into how developers use these terms to express different types of meaning. For RQ2, we investigate how closed-category terms correlate with three types of context: (1) the local programming context in which a variable is declared (e.g., Function, Attribute), (2) the programming language of the source code in which the identifier was found, and (3) the broader system-level domain of the software in which it appears (e.g., domain-specific vs general-purpose projects). This 3-way perspective allows us to examine both how these terms are used *within* individual source code structures, between programming languages, and how they reflect distinctions *across* different kinds of systems.

We begin by analyzing the distribution of four closed categories: prepositions, determiners, conjunctions, and numerals, across five programming contexts and three programming languages. We discuss which categories are most frequent in which contexts/languages and consider how those patterns may reflect the communicative goals of the developer. We then extend this analysis to system-level

Table 9: Results of Pearson's Chi-Squared Test. df = 6, $\alpha = 0.05$, critical value = 12.592, test statistic = 4.291

	C	C++	Java	Chi-square per row
D	0.451887	0.275300	1.282415	2.009603
DT	0.457607	0.036988	0.678404	1.172999
P	0.056554	0.015121	0.116729	0.188405
CJ	0.621604	0.157629	0.140876	0.920108
Chi-square per column	1.587652	0.485038	2.218424	4.291115

domain context, comparing the normalized frequency of closed-category term usage between domain-specific and general-purpose systems.

6.1 Closed-Category Term Usage Across Programming Contexts, Programming Languages, and System Domains

We now examine how differing contexts and closed-category grammar patterns relate to one another, and whether programming language further conditions their usage. We begin by analyzing cross-language correlations in the usage of closed-category terms, followed by an exploration of correlations in how these terms are used across different program contexts. We provide Table 8, which shows frequencies and percentages for PoS and terms, to help the reader understand what types of terms are most prevalent. However, for this research question, we rely primarily on Tables 9, 10, 11, and 12, which present the results of our Pearson Chi-square tests and standardized Pearson residuals. Using these, we highlight common patterns, terms, and the contexts or languages to which these patterns are correlated.

6.1.1 ***Language-Specific Differences in Closed-Category Term Usage***

Starting with an analysis of closed category terms and programming language, our **null hypothesis** is that there is no relationship between identifiers containing closed category terms and the programming language in which they appear. Our **alternative hypothesis** is that there is a relationship between identifiers that contain closed category terms and the programming language.

Methodology. To perform our Chi-Square test, we use the CCID described in Section 4.1. We count how many times each closed-category PoS appears in C++, Java, or C code by analyzing all 1,001 identifiers that contain closed-category terms. For example, we might find that there were 20 numerals in our data set found in C++ code, and 5 numerals in Java code. Once we have these frequencies, we apply the Chi-Square test and Standardized Pearson residuals with Bonferroni correction to determine overall significance and per-part-of-speech significance, respectively.

Results. The Chi-square test for programming language (Table 9) did not produce a statistically significant result. Thus, we do not reject the null hypothesis: **there is no strong evidence that closed-category tag usage differs significantly by programming language**. However, exploratory analysis of the Standardized Pearson residuals in Table 10 offers insight into modest trends worth noting:

Table 10: Standardized Pearson Residuals Results. With Bonferroni Correction, a significant result is $\alpha = 0.05/12 = 0.0042$, which translates to a ± 2.87 critical value

	C	C++	Java
D	0.953270	0.740778	-1.649081
DT	-0.986465	-0.279221	1.233406
P	-0.367728	-0.189311	0.542514
CJ	0.983056	-0.492859	-0.480581

Table 11: Results of Pearson's Chi-Squared Test. df = 12, $\alpha = 0.05$, critical value = 21.026, test statistic = **88.893567**.

	ATTRIBUTE	CLASS	DECLARATION	FUNCTION	PARAMETER	Chi-square per row
D	0.44932	16.031139	0.776138	15.916173	10.634467	43.807237
DT	0.511266	6.398601	2.14863	0.959251	0.171805	10.189553
P	0.332388	0.506133	3.498335	8.724009	3.63817	16.699033
CJ	3.366551	1.027972	0.233783	12.071429	1.498009	18.197744
Chi-square per column	4.659525	23.963845	6.656886	37.670861	15.942451	**88.893567**

Table 12: Standardized Pearson Residuals. With Bonferroni correction, a significant result is $\alpha = 0.05/20 = 0.0025$, which translates to a ± 3.02 critical value.

	ATTRIBUTE	CLASS	DECLARATION	FUNCTION	PARAMETER
D	-0.905532	**4.719156**	1.1768	**-5.505053**	**4.254103**
DT	0.993307	**-3.065907**	2.013483	-1.389769	-0.556035
P	0.849262	-0.91434	-2.724314	**4.444202**	-2.713221
CJ	-2.179409	-1.050735	-0.567884	**4.21543**	-1.403873

- **Numerals (D)** are modestly underrepresented in **Java** (residual = –1.65), suggesting a mild tendency to avoid numeric suffixes in Java naming.
- **Determiners (DT)** are slightly overrepresented in **Java** (residual = 1.23), potentially reflecting more frequent use of quantifying or contextual modifiers.

Summary. While we did not find significant statistical evidence linking closed-category tag usage to programming language, the residual analysis and qualitative trends suggest mild idiomatic differences, particularly around numeral usage and determiner phrasing. For example, in our Axial Code data from RQ1, **Population/Subpopulation Reference** identifiers were found in Java (21, 50%) and C++ (18, 42%) more than in C(3, 7%). These patterns may reflect broader stylistic conventions or design idioms of each language, but should be interpreted cautiously given the statistical outcome.

6.1.2 Context-Specific Differences in Closed-Category Term Usage

Next, we analyze the correlation between closed-category terms and program contexts such as Function names, Attributes, and Class names. Our **null hypothesis** is that there is no relationship between identifiers containing closed-category terms and the context in which they appear. Our **alternative hypothesis** is that there is a relationship between identifiers containing closed-category terms and the context in which they appear.

Methodology. To perform our Chi-Square test, we use the CCID described in Section 4.1. We analyzed all 1,001 identifiers that contained closed-category terms, and counted how many times a closed-category term appears in one of our five code contexts: Attribute, Function, Class, Declaration, or Parameter. Once we have these frequencies, we apply the Chi-Square test and Standardized Pearson residuals with Bonferroni correction to determine overall significance and per-part-of-speech significance, respectively.

Results. The Chi-squared test for context (Table 11) shows a significant result ($88.89 > 21.026$), allowing us to **reject the null hypothesis**. As before, we analyze the Standardized Pearson Residuals (Table 12) to understand where the largest deviations appeared.

Conjunctions (CJ). Closed-category grammar patterns that include conjunctions typically feature the terms 'and', 'or', or 'if', as reflected in Table 8. Although rare overall, these patterns are significantly positively correlated with function names (Standardized Pearson residual = 4.22, Table 12). This indicates that when conjunctions do appear, they are far more likely to occur in function names than in other contexts.

The selective coding data from RQ1 explains this pattern. Conjunction-based grammar patterns tend to express compound logic, dual-purpose behavior, or guarded activation, which are most relevant when naming behaviors or actions rather than static values. For example, **Guarded Action / Conditional Enablement** patterns such as load_if_needed or activate_if_enabled appear in function names to encode preconditions or gating logic directly into the identifier.

Conjunction-based names are largely absent from declarations and classes, likely because those contexts do not typically represent conditional or compound operations. The data supports the interpretation that developers strategically use conjunctions in function names to foreground complex control logic or behavioral nuance at the point of execution.

While we did find a significant correlation between conjunctions and functions, it is essential to note that our dataset contains 50 conjunctions, meaning that while we have identified potential trends, further research on a larger sample is likely necessary.

Determiners (DT). Closed-category grammar patterns that include determiners typically feature the terms `last`, `next`, `all`, `no`, or `this`, as shown in Table 8. While determiners are not significantly positively correlated with any specific context, they are modestly negatively correlated with class names (Standardized Pearson residual = –3.07, Table 12), suggesting that developers tend to avoid determiner-based grammar patterns in class names.

The selective coding analysis offers a plausible explanation: the most common roles for determiners involve expressing temporal or positional relationships—such as **Temporal / Most Recent Element** and **Temporal / Upcoming Element** (over 100 instances)—as well as set-based semantics, such as **Population / Subpopulation Reference** (38 instances). These patterns commonly use terms like `next`, `last`, `prev`, `all`, and `any` to indicate an element's position in a sequence or its membership in a filtered subset.

These naming strategies are well-suited to attributes, parameters, and declarations, where variables often represent dynamic state or bounded subsets. In contrast, class names are generally used to describe abstract data types or roles,

where positional or filtering semantics are less relevant. The relative absence of determiners in class contexts thus reflects their semantic focus: determiners foreground state, scope, or specificity, whereas class names typically signal structural purpose or generalization.

Numerals (D). Closed-category grammar patterns that include numerals often feature numerals such as `1`, `2`, `0`, `3`, and `4`, as shown in Table 8. Numerals are significantly positively correlated with parameter names and class names, and significantly negatively correlated with function names (Standardized Pearson residual = 4.72 for class names, 4.25 for parameters, and –5.51 for functions; Table 12). Notably, numerals are the only closed-category category to exhibit a positive correlation with class names.

The selective coding data sheds light on this trend. The most frequent numeral-related patterns in our dataset fall under **Distinguisher × Human-Named Convention** and **Distinguisher × Locally Specific Concept**. These naming strategies are used to distinguish among similar entities (e.g., `arg1`, `arg2`, `tile3`) or to embed system-specific references (e.g., `m34`, `cp437`) into variable or type names. Such distinctions are especially useful in parameters and declarations, where there is no syntactic support for disambiguation outside of naming.

In contrast, function-level disambiguation is often handled by the language itself, through overloading, polymorphism, or naming conventions focused on behavior; making numerals largely unnecessary or even undesirable in that context. Their absence from function names reflects this shift: numerals encode identity, that is, they serve as a means of traceability to specific domain concepts and distinguish entities with similar names, rather than encoding purpose or behavior.

Taken together, these findings suggest that numerals serve primarily as disambiguators or protocol markers rather than communicative devices for expressing behavior. Their presence in class and parameter names signals static or structural variation, while their avoidance in function names underscores a developer's preference for meaningful, descriptive action labels over numerical markers.

Prepositions (P). Closed-category grammar patterns that include prepositions frequently feature terms such as `to`, `for`, `as`, `on`, or `from`, as shown in Table 8. These patterns are significantly positively correlated with function names (Standardized Pearson residual = 4.44, Table 12), suggesting that developers are particularly likely to use prepositional grammar when naming behaviors or operations.

This strong correlation reflects the behavioral semantics that prepositions convey in identifier names. As detailed in our selective coding, prepositions frequently express directionality, transformation, conditional activation, or event-driven execution; all of which are highly function-oriented behaviors, requiring action to be taken.

Summary. The results of our analysis support the alternative hypothesis: closed-category parts of speech are meaningfully correlated with specific roles and contexts in source code. Prepositions and conjunctions appear more frequently in function names, where they help express behavioral nuances such as guarded actions, type casting, or alternative execution paths. Numerals, by contrast, are most commonly found in class names and parameter declarations, where they signal disambiguation, indexing, or versioning; identifiers rooted in identity rather than behavior.

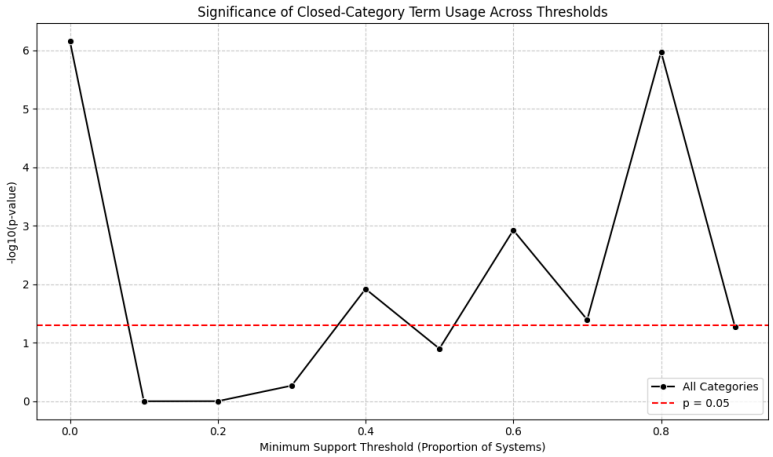

Fig. 2: Global Mann-Whitney U test significance across thresholds, showing divergence between domain-specific and general systems. Peaks at 0.6 and 0.8 suggest the importance of both ubiquitous and moderately specific closed-category terms.

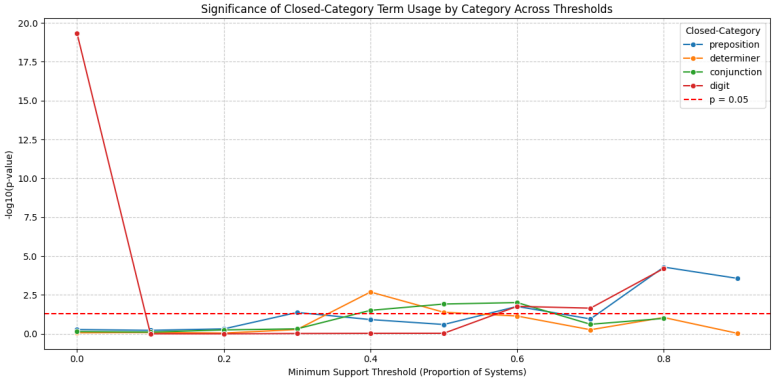

Fig. 3: Per-category Mann-Whitney U test significance across thresholds. Prepositions dominate across thresholds, while conjunctions and numerals contribute more variably.

6.1.3 ***Closed-Category Term Usage Across System Domains***

Having established correlations between closed-category terms, source code context, and programming language, we now turn to a broader question: do these terms also vary with the domain of the software system itself? This sub-question allows us to further test our central hypothesis, that closed-category terms are not used arbitrarily, but instead reflect domain-relevant distinctions in how behavior and structure are communicated. If certain domains make more frequent or specialized use of closed-category terms, this suggests that such terms play a role in expressing concepts tightly coupled to those domains. Understanding and appropriately using these terms may therefore be critical for accurate communication of behavior in domain-specific software.

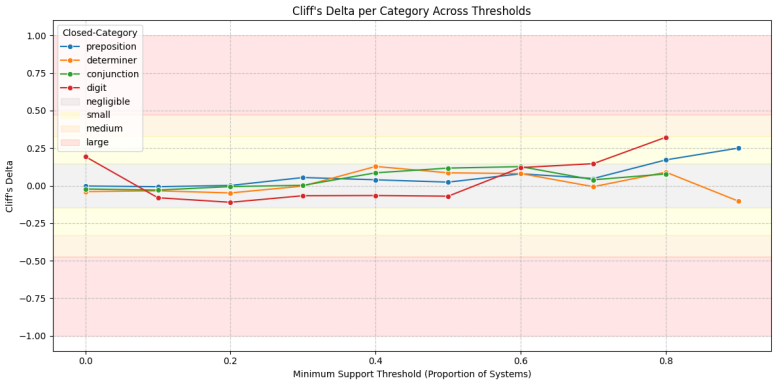

Fig. 4: Cliff's Delta for closed-category terms across system support thresholds.

Table 13: Systems and System Domains Selected Based on Axial Codes for each Closed-Category Type

Closed-Category Type	Axial Code	Domain	GitHub URL	Language
Prepositions	Type Casting / Interpretation	Serialization/Deserialization Libraries	https://github.com/msgpack/msgpack-c/	C
Prepositions	Type Casting / Interpretation	Serialization/Deserialization Libraries	https://github.com/open-source-parsers/jsoncpp.git	C++
Prepositions	Type Casting / Interpretation	Polyglot Interop Tools / Type Bridge Layers	https://github.com/pybind/pybind11	C++
Prepositions	Type Casting / Interpretation	Polyglot Interop Tools / Type Bridge Layers	https://github.com/swig/swig	C++
Determiners	Position in Sequence (Temporal / Recent & Upcoming)	Parser Generators / Token Stream Libraries	https://github.com/akimd/bison	C
Determiners	Position in Sequence (Temporal / Recent & Upcoming)	Parser Generators / Token Stream Libraries	https://github.com/antlr/antlr4	Java
Determiners	Position in Sequence (Temporal / Recent & Upcoming)	Job Queues / Schedulers	https://github.com/PerMalmberg/libcron	C++
Determiners	Position in Sequence (Temporal / Recent & Upcoming)	Job Queues / Schedulers	https://github.com/quartz-scheduler/quartz	Java
Prepositions	Position / Ordering in Time or Space	Data Structure / Algorithm Libraries	https://github.com/apache/commons-collections	Java
Prepositions	Position / Ordering in Time or Space	Data Structure / Algorithm Libraries	https://github.com/boostorg/container	C++
Prepositions	Position / Ordering in Time or Space	Compiler / Intermediate Representation Tools	https://github.com/rose-compiler/rose	C
Prepositions	Position / Ordering in Time or Space	Compiler / Intermediate Representation Tools	https://github.com/TinyCC/tinycc	C
Determiners	Population / Subpopulation Reference	Dataframe / Matrix Libraries	https://github.com/apache/arrow	C++
Determiners	Population / Subpopulation Reference	ML Preprocessing / Feature Selection	https://github.com/haifengl/smile	Java
Determiners	Population / Subpopulation Reference	Dataframe / Matrix Libraries	https://github.com/jtablesaw/tablesaw	Java
Determiners	Population / Subpopulation Reference	ML Preprocessing / Feature Selection	https://github.com/pjreddie/darknet	C
Conjunctions	Guarded Action / Conditional Enablement	UI Libraries / Event Dispatch Systems	https://github.com/GNOME/gtk	C
Conjunctions	Guarded Action / Conditional Enablement	Feature Flag Systems / Config-Driven Execution	https://github.com/lightbend/config	Java
Conjunctions	Guarded Action / Conditional Enablement	UI Libraries / Event Dispatch Systems	https://github.com/ocornut/imgui	C++
Conjunctions	Guarded Action / Conditional Enablement	Feature Flag Systems / Config-Driven Execution	https://github.com/spring-projects/spring-boot	Java
Digits	Distinguisher × Locally Specific Concept	Game Engines / Grid-based Games	https://github.com/godotengine/godot	C++
Digits	Distinguisher × Locally Specific Concept	Game Engines / Grid-based Games	https://github.com/jMonkeyEngine/jmonkeyengine	Java
Digits	Distinguisher × Locally Specific Concept	Scientific Computing / Matrix Libraries	https://github.com/OpenMathLib/OpenBLAS	C
Digits	Distinguisher × Locally Specific Concept	Scientific Computing / Matrix Libraries	https://github.com/PX4/eigen	C++
Digits	Distinguisher × Human-Named Convention	Code Generators / Macro Frameworks	https://github.com/jhipster/jhipster-bom	Java
Digits	Distinguisher × Human-Named Convention	GUI Builders / Form Designers	https://github.com/qt/qtbase	C++
Conjunctions	Data Pair / Composite Value	Multi-format I/O Libraries	https://github.com/apache/parquet-java	Java
Conjunctions	Data Pair / Composite Value	Cryptographic Libraries	https://github.com/jedisct1/libsodium	C
Conjunctions	Data Pair / Composite Value	Multi-format I/O Libraries	https://github.com/libjpeg-turbo/libjpeg-turbo	C
Conjunctions	Data Pair / Composite Value	Cryptographic Libraries	https://github.com/openssl/openssl	C

Methodology. In RQ1, we developed a set of Axial Codes to describe the behavioral roles of closed-category terms in identifiers. To explore their importance at the level of system domain, we selected the two most common Axial Codes from each closed-category group (e.g., Prepositions, Determiners). For each code, we identified two software domains that we hypothesized would frequently use identifiers expressing that behavior. For example, in the Preposition group, the top two Axial Codes were:

- **Type Casting / Interpretation**
- **Position / Ordering in Time / Space / Execution Context**

Based on these, we selected four relevant software domains:

- For **Type Casting / Interpretation**:

  - Serialization/deserialization libraries
  - Polyglot interop tools or type bridge layers
- For **Position / Ordering in Time / Space / Execution Context**:
  - Data structure and algorithm libraries
  - Compiler or intermediate representation (IR) tooling

Table 13 lists all selected systems, the domain they represent, and the Axial Code that motivated their inclusion. To fit the table, we omitted a few details like system size; this information can be found in our open data set (Section 11.5). We analyzed identifiers drawn from five programming contexts—attributes, parameters, functions, declarations, and class names—across two groups of systems: one curated for domain-specific relevance and one composed of general-purpose projects, which were used to construct the CCID (Table 3). The general-purpose group serves as a baseline, as these systems were not selected based on any particular domain hypothesis. Our underlying assumption is that, if closed-category terms are meaningfully correlated with domain-specific concerns, we will observe statistically significant differences in their usage between these two groups.

For each system, we extracted all identifiers and segmented them using Spiral [42]. We then filtered out all terms that are neither numerals nor included in our predefined lists of closed-category terms (as defined in Section 4). After filtering, we compute the normalized frequency of closed-category term usage by dividing the count of qualifying terms by the system's total lines of code. To assess whether the differences in usage were statistically significant, we applied a Mann-Whitney U test to compare the distributions between domain-specific and general-purpose systems.

Results. To mitigate the risk of a small number of systems dominating the term distribution, and to better understand how widely closed-category terms are used, we introduce a support threshold that controls how many systems a term must appear in to be included in the Mann-Whitney U test. Increasing the threshold emphasizes more widely used (*ubiquitous*) closed-category terms; decreasing it emphasizes more narrowly distributed (*specific*) closed-category terms that may signal domain-specific behavior. Significance at high thresholds implies that there are terms that are important to all of our domain-specific systems; significance at lower thresholds implies that there are subsets of the domain-systems that make use of terms that are not very universal, but nevertheless set these systems apart from the general set.

To explore how these different usage profiles affect our results, we conducted a threshold sweep. At each level, a term had to appear in at least a given proportion of systems to be retained. This allowed us to systematically vary our emphasis between ubiquity (terms common across many systems) and specificity (terms concentrated in a smaller, domain-aligned subset). The results, shown in Figure 2, reveal the most substantial distributional divergence at thresholds around 0.6 and 0.8. These peaks suggest that both common and moderately specific terms help distinguish domain-specific systems. By contrast, thresholds between 0.1 and 0.3 yielded little significance, likely reflecting linguistic noise from terms with low usage or ambiguous semantic function.

We repeat the analysis at the level of individual closed-category types (prepositions, determiners, conjunctions, numerals) to identify which groups drive the observed differences. As shown in Figure 3, prepositions exhibit consistently strong

significance across thresholds, particularly between 0.6 and 0.8. Numerals and conjunctions show more variable but still notable divergence, while determiners contribute the weakest and least consistent signal. These trends suggest that domain-specific systems rely more heavily on certain linguistic forms, especially prepositions, to express structural or behavioral distinctions central to their design.

To complement the significance testing, we examine *Cliff's Delta* as a non-parametric effect size estimate, plotted in Figure 4. This allows us to assess not only whether closed-category usage differs between system types, but also how strongly. The results show that **prepositions** and **numerals** increasingly favor domain-specific systems at higher thresholds, reaching small to medium effect sizes. **Determiners**, by contrast, exhibit weak or even negative effect sizes, suggesting a more general-purpose usage profile. **Conjunctions** remain close to the negligible–small range, with mild domain skew. These patterns reinforce the idea that domain-specific systems do not just differ in which closed-category terms they use, but in how salient those terms are among their most widely reused identifiers.

Summary. Our findings suggest that domain-specific systems tend to use closed-category terms more frequently than general-purpose baselines, particularly in ways that align with the communicative roles captured by our Selective Codes. While we rely on predefined lists of closed-category terms—without verifying each term's function in context—our goal in this evaluation was not to establish definitive usage, but to assess whether these terms might play a heightened role in domain-specific software. The statistical results support that possibility. As such, we argue that further research into how closed-category terms contribute to domain-specific expression is both warranted and promising. These findings offer initial evidence that supporting developers in the effective use of such terms could benefit certain styles or domains of software development.

6.2 Summary of RQ2

For RQ2, we examine how closed-category terms correlate with multiple forms of context: (1) source-code-local structure, (2) programming language, and (3) broader system domain. Our findings reveal several consistent trends. First, there is no statistically significant difference in the distribution of closed-category terms across the three programming languages under study, though there are some trends that indicate how they may differ in minor (i.e., non-statistically-significant) ways. Second, source code context plays a significant role: Prepositions and conjunctions are used disproportionately in function names; numerals are **significantly positively** correlated with parameters and class names while **significantly negatively** correlated with function names; and Determiners are significantly negatively correlated with class names. These patterns align with the communicative roles uncovered in our Selective Codes, such as the use of prepositions to express behavior or data flow, and numerals to distinguish instances or versions.

Finally, we found statistical evidence that domain-specific systems use closed-category terms more frequently than general-purpose ones. This suggests that these terms serve as meaningful signals of domain-relevant behavior. Taken together, our results demonstrate that **closed-category terms have specific, purposeful usage in software development**.

One of the broader aims of RQ2 is to assess whether closed-category terms are meaningful enough to warrant a dedicated study. We argue that their statistically significant correlations with specific code contexts support the need for further research: such terms appear deliberately and consistently in ways that reflect their natural language functions. While our domain-level comparison relies on predefined lists of closed-category terms, without manual verification of each term's grammatical role, the results nonetheless suggest that these terms may hold particular communicative importance in domain-specific software. Supporting the appropriate use of closed-category terms through tools, naming conventions, or educational interventions may ultimately benefit program comprehension and internal quality, particularly in domains where such terms help convey behavioral intent.

7 Related Work

While numerous studies have been conducted on identifier names, this paper represents one of the few to address closed-category terms, and the only paper to conduct an in-depth analysis of their usage in open-source systems. We discuss relevant related literature below, and how our work can be improved by, or improve upon, their outcomes.

7.1 Grounded Theory in Software Engineering

Our methodology is inspired by Straussian grounded theory [22], which emphasizes iterative coding, constant comparison, and the development of conceptual categories grounded in data. While we drew inspiration from classic Straussian grounded theory, our approach adapts it to the structure of source code, treating identifiers as theory-generating artifacts rather than unstructured text or human narratives. We employed several key components of the method, including open coding, memoing, axial coding, selective coding, and constant comparison, to analyze identifier names in the context of surrounding code and comments.

Our analysis reached theoretical saturation in that we developed axial codes iteratively until they accounted for all observed behaviors, refining them as new cases emerged. We also engaged in a form of core category development through cross-category synthesis, identifying broader trends that link behavioral interpretations across different grammatical constructs such as determiners, prepositions, digits, and conjunctions.

As grounded theory was originally developed in sociology, its application to the semi-structured nature of software artifacts, including lexical constructs like variable names, presents unique challenges. Some of these challenges have been explored in prior work on grounded theory in software engineering [3,35,71], which informed our adapted approach. Our methodology is an example of how grounded-theory-based techniques can support the analysis of naming practices, particularly when grammar patterns and part-of-speech information are involved.

7.2 Part of Speech Taggers

POSSE [31] and SWUM [33], and SCANL tagger [57] 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) [31, 33]. They leverage this information to help determine what POS to assign different words. Newman et al. [55] compared these taggers on a larger dataset than their original evaluation (1,335 identifiers) using five identifier categories: function, class, attribute, parameter, and declaration statement. They found that SWUM was the most accurate overall, with an average accuracy around 59.4% at the identifier level. Later, Newman et al. created a new tagger that ensembled SWUM, POSSE, and Stanford together, then compared with SWUM, POSSE, and Stanford [73] individually, finding that the ensembled tagger exceeded the others' performance metrics on identifiers [57].

7.3 Human-subjects studies

Several studies use human subjects to understand the influence and importance of different characteristics of identifiers. Our work is largely complementary to these studies, as it can be used in conjunction with data from these studies to create/support naming techniques. Reem et al. [6] conducted a survey of 1100 professional developers, shedding light on developer preferences and practices regarding the content of identifier names, including the use of abbreviations and preferred identifier length. Feitelson et al. [26] studied how the information content of identifiers named affected their memorability, and concluded that short names that contain focused information are likely optimal. Felienne et al. [74] find, among other things, that while instructors agree on the importance of naming, there is disagreement between their teaching practices. Even internally, teachers are generally inconsistent in how they teach and practice identifier naming in the classroom. The results of their study highlight the importance of increasing our formal understanding of naming, which can help grow and support the consistency of teaching materials and practices in the classroom.

7.4 Rename Analysis

Arnoudova et al. [10] 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 anti-patterns [9]. Liu et al. [46] 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 [47]. Peruma et al. [62] 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 [63] and data type changes [64] that co-occur with renames. Osumi et al. [59] studied terms that were co-renamed with a goal of supporting developers in deciding when identifiers should be renamed together. In particular, they studied how location, data dependencies, type relationships, and inflections affected co-renaming.

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 complementary; our analysis of naming structure can be used to help improve how these techniques analyze changes between two versions of a name by examining changes in their grammar pattern. In particular, since we specifically study closed-category terms, rename analysis can leverage our results to improve its behavior on identifiers that contain these terms. For example, they might use our results to determine when to recommend a closed-category term during a rename operation.

7.5 Identifier Type and Name Generation

There are many recent approaches to appraising identifier names for variables, functions, and classes. Kashiwabara et al. [43] use association rule mining to identify verbs that might be good candidates for use in method names. Abebe [2] uses an ontology that models the word relationships within a piece of software. Saeed et al. [60] vectorize methods based on metrics and use the K-Nearst Neighbors algorithm with these vectors, and a large data set of methods, to recommend method names. Allamanis et al. [5] introduce a novel language model called the Subtoken Context Model. There has also been work to reverse engineer data types from identifiers [32, 49]. 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 (or understand their behavior for the purpose of generating types). There is a lot of work in this subfield, but the contrast to our work remains the same for all of them: These approaches aim to predict strong identifier names based on history. Our approach can help, since an understanding of common naming structures can support filtering out names that are inappropriate based on their grammatical structure; teach AI-based approaches how to optimize the identifiers they generate, or at least avoid using bad grammar structure; or help reverse-engineer the semantics of an identifier name based on its grammatical properties. In addition, automated name generation approaches cannot teach us much about naming practices on their own, nor can they help us formalize our understanding of strong naming structures and how those can be taught in a classroom. Thus, our work is novel, and complementary to identifier name generation approaches.

7.6 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 [1, 24, 28, 66, 67]. Abebe and Tonella [1] 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 [51] 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 [67] 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 metamodel [24] and a second paper on linking programs to ontologies [66]. 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.

Many software word ontologies use meta-data about words to understand the relationship between different words. There is a synergistic relationship between the work presented here and software ontologies, as stronger ontologies can facilitate the effective generation and study of grammar patterns, and the CCID can aid in constructing stronger software word ontologies. In particular, studying closed-category terms helps strengthen the metadata used to generate an ontology that seeks to map how words are related to one another, or code behavior.

7.7 Identifier Structure and Semantics Analysis

Liblit et al. [45] discuss naming in several programming languages and make observations about how natural language influences the use of words in these languages. Schankin et al. [68] focus on investigating the impact of more informative identifiers on code comprehension. Their findings show the advantage of descriptive, compound identifiers over short single-word ones. Hofmeister et al [36] 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. [44] 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 [16] extends their previous work on Java class identifiers [15] 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., PoS).

Caprile and Tonella [19] analyze 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 used the classes identified in this previous work to propose methods for restructuring program identifiers [18]. Fry and Shepherd [29,69] 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).

Høst and Østvold study method names as part of a line of work discussed in Høst's dissertation [38]. 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 [37]. 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 [40] 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 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 similar to the general grammar patterns studied in our prior work [55]. They extend this use of phrases by presenting an approach to debug method names [39]. 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 PoS tags (i.e., phrases) and concretizes 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 [41], Høst and Østvold analyzed how ambiguous verbs in method names makes comprehension of Java programs more difficult. They proposed a method to detect when two or more verbs are synonymous and used to describe the same behavior in a program, aiming to eliminate these redundancies while increasing 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.

Butler [17] 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 [70], who identified a grammar pattern for class names of (adjective)* (noun)+ and studies how class names correlate with micro patterns. Among 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 [14]. 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. Olney [58] compared taggers for accuracy on identifiers, but only on Java method names which were curated to remove ambiguous words (e.g., abbreviations).

Binkley et al [12] studied 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". Al Madi [4] created a tool for performing lexical analysis of identifier names based on phonological, semantic, and orthographic similarity. Techniques that normalize identifiers, such as the one presented by Jingxuan [75], or by Hill [34] can help make generating grammar patterns easier by expanding abbreviations into full words that a tagger can recognize more accurately. Aman et al. [8] studied confusing variable pairs, which are variables with very similar names, to understand how/if they are changed over time, and how pervasive they are.

None of the projects in this subsection deal specifically with closed-category grammar patterns, or even terms that fall within a closed PoS category. Many of them, particularly the work on PoS taggers, on grammar patterns in differing contexts, normalizing identifier names, and on grammatical anti-patterns, are likely mutually-synergistic to our work. This is because a stronger understanding of closed-category terms/patterns, and how they relate to program behavior, can help support the style of analysis these works leverage.

8 Discussion and Future Work

Closed-category terms—including determiners, prepositions, conjunctions, and numerals—are compact grammatical structures that carry dense behavioral meaning in source code. Developers use them to embed logical cues, control flow, and intent directly into identifiers. Based on our findings, we outline several actionable implications for tool builders, educators, and developers:

1. **Treat closed-category terms as lightweight cognitive annotations.** Developers should consider how they might improve identifier clarity by deliberately using closed-category terms to signal concepts such as selection (`allItems`), temporality (`nextNode`), negation (`noCache`), disjunction (`keyOrIV`), or encode system-specific roles/concepts (e.g., `arg1`, `Neo4j`). These terms encode behavioral roles that may be missed in names relying solely on open-class words. *Naming checkers, LLMs, or documentation generators should highlight or recommend these roles to aid consistency and reader comprehension, as well as assist developers in reflecting on how they might improve their terminology usage in identifiers. Our axial codes offer a schema that could seed such tools.*
2. **Augment use of closed-category terms by considering code context.** Our data show that closed-category term usage varies systematically across code contexts. For example, determiners like `no`, `next`, and `this` are very unlikely to appear in class names, while prepositions such as `on` or `to` are common in function names but rare elsewhere. *As developers consider when to use closed category terms in their coding, they can leverage context to help in their decision making. Tool support can make this easier– IDEs and linters should offer*

context-aware naming prompts, e.g., flagging uncommon use of conjunctions in class names or suggesting appropriate determiners for declarations.

3. **Use grammar patterns to reveal relational structure.** Closed-category terms not only express behavior, they also help structure it. Patterns like `DT NM N` and `P NM N` typically signal unary relations (e.g., `lastError`, `asFloat`), while `N CJ N` or `N P N` often signal binary ones (e.g., `dataOrLogger`, `readFromDisk`). Function names with final prepositions (e.g., `sendTo`) sometimes encode a second operand passed via parameters. *Developers should consider the latent logical structure of the grammatical patterns they use when naming. Static analyzers or naming tools could infer missing operand roles or flag relational ambiguity. For example, encountering a binary-looking pattern in a context that provides only one operand could trigger a naming prompt.*

These findings have practical implications for tool builders and educators interested in improving naming support. For example, grammar patterns could be integrated into static analysis or IDE plugins to provide optional, contextual suggestions; highlighting when an identifier follows an uncommon structure or when the pattern contrasts with its code context. This does not imply the name is incorrect, but it may prompt a developer or reviewer to reflect on whether the chosen pattern aligns with the intended semantics. For instance, encountering a pattern like `N CJ N` (e.g., `dataOrLogger`) in a constant declaration could trigger a soft prompt: "This naming pattern is rare in this context—consider whether it clearly communicates its role."

Grammar patterns can also be used to scaffold naming suggestions in LLM-driven tools. Instead of generating identifiers purely from task descriptions, models could be prompted to produce names that instantiate common open- or closed-category structures (e.g., `DT NM N`, `P N`), which are frequently associated with behavioral semantics. This approach may help align completions with human naming conventions, while still allowing flexibility in term choice.

Beyond tooling, grammar pattern awareness can enhance educational workflows. Instructors could use pattern frequency and semantics to illustrate naming "idioms," helping students understand how experienced developers encode behavior through compact syntactic forms. Similarly, code review tools might use grammar pattern summaries to draw attention to unconventional naming constructs, offering reviewers an additional signal without enforcing rigid standards.

In future work, we plan to evaluate these ideas empirically: measuring whether adherence to common patterns improves comprehension, how grammar scaffolding affects naming quality in LLM-generated identifiers, and whether tools that surface grammar patterns can meaningfully assist developers. In addition, it would be interesting to perform studies simialar to Schankin [68] and Hofmeister [36]'s work on how the descriptiveness of names influence comprehension; or Arnaoudova [9], and Host [39] who looked at how code behavior and naming structure could be used to measure name quality. Specifically, we could study how closed-category terms change or augment the outcomes of their studies.

9 Threats to Validity

Construct Validity: This study is conducted on a manually annotated dataset of 1,275 identifiers containing closed-category terms, the largest of its kind at the time

of writing. A potential threat lies in the completeness of our closed-category term list: we relied on a predefined lexicon (Section 4), meaning novel or unlisted terms may be absent from the dataset. However, their absence would likely expand our results rather than refute them. Our identifier sample is restricted to production code, and although we excluded known test files, developers may occasionally include test logic in production files. To mitigate this, we manually reviewed each identifier and its source context. Furthermore, while we used file extensions to distinguish C (.c, .h) from C++ (.cpp, .hpp), these conventions are not absolute. We addressed this by manually validating the source language of each identifier.

Internal Validity: Abbreviations within identifiers were not expanded, which may have caused occasional misinterpretation by annotators. However, annotators had access to the surrounding source code, reducing the risk of misannotation. Grammar pattern tagging and axial coding for each closed-category term were both subject to cross-validation by three independent annotators and evaluated using Fleiss' Kappa to assess agreement. We used a grounded-theory-inspired approach to develop our behavioral codes. Four coders participated in open and axial coding; during the selective coding phase, one coder proposed all selective codes, which were then validated and refined collaboratively by the other three through discussion until thematic saturation was reached.

We used statistical methods to examine correlations between closed-category terms and contextual variables. We performed two chi-square tests: one to assess correlation between closed-category part-of-speech categories (e.g., Determiner, Preposition) and programming language (Java, C, C++), and another for their correlation with code context (Attribute, Function, Declaration, Parameter, Class), derived automatically via srcML [20]. We applied Bonferroni correction to account for multiple comparisons. A threat to internal validity is the assumption of independence in the chi-squared test. If violated, some significant values may be distorted. However, the primary insights of RQ1, which focus on behavioral coding through qualitative analysis, are unaffected by this statistical assumption.

External Validity: Our data includes identifiers from C, C++, and Java, three widely used languages with similar syntactic and object-oriented paradigms. While this helps reduce language-specific bias, our findings may not generalize to other paradigms such as functional or logic-based languages, where naming conventions and code contexts may differ significantly.

Mitigation Strategies: To ensure transparency and reproducibility, the dataset will be made publicly available (Section 11.5). Annotators were allowed to inspect source code when labeling identifiers, and each identifier was independently annotated twice. Grammar patterns and axial codes were validated by multiple annotators, with inter-rater agreement assessed using Fleiss' Kappa. We selected a representative sample from 30 software systems, sized to meet a 95% confidence level with a 5% confidence interval. Code context was derived automatically using `srcML`. Finally, to evaluate whether closed-category term usage varies by domain, we curated a domain-specific dataset (e.g., compilers, databases, networking tools) and compared it against a general-purpose set selected without regard to domain. We applied a Mann-Whitney U test to compare term frequencies between these groups, normalizing by lines of code to control for system size.

10 Conclusions

This paper presents a detailed empirical study of **closed-category terms in identifier names**, highlighting how they function semantically across a wide range of software artifacts. Our contributions include:

1. **The CCID dataset:** A new, part-of-speech–annotated dataset of identifier names containing closed-category terms, released with this paper. It supplements prior datasets focused on open-class lexical items, enabling more nuanced research into naming semantics.
2. **A dual-level coding framework:** We introduce a combined selective and axial coding scheme to interpret the semantics of determiners, prepositions, conjunctions, and numerals in identifiers. This framework maps grammar structure to conceptual behavior in a way not previously formalized, and provides a solid basis for future research and development of naming practices.
3. **Insights into the distinct semantic roles of closed-category words:** Our study shows how programming diverges from natural language in its use of terms like `next`, `both`, and `or`, and how these terms reflect functional intent embedded in code structure.

These findings have practical implications for code review, refactoring, and intelligent naming support. Automated tools can leverage our work to suggest context-appropriate names, detect inconsistent patterns, or provide just-in-time feedback during the development process. The insights we have uncovered can be packaged and curated by educators to teach semantic clarity in naming, moving beyond generic best practices to behavior-specific guidance.

We also outline several promising directions for future research:

1. Further validation of the behavioral categories we propose, especially through comprehension studies that measure how different naming structures affect developer understanding. This could also extend to examining the correlation between the beahvoral categories and software quality metrics.
2. Experimental work to test which closed-category naming patterns improve or hinder program comprehension, while controlling for code context and developer experience.
3. Integration of our findings into intelligent development tools, including naming recommendation systems, refactoring support, and automated code reviewers capable of detecting semantic mismatches or naming anti-patterns.

This work demonstrates that closed-category terms are not linguistic noise; they are deliberate, behaviorally meaningful tools in the software naming arsenal. By mapping their roles across grammar patterns and program contexts, we provide a foundation for future studies in naming semantics and for tools that support naming literacy. We believe our findings raise the question: **Should closed-category terms be used more often in naming?** It is clear that they are used purposefully, and we believe that their use does help comprehension, but in which situations is that true? And how can we detect when such terms should be used, as opposed to another naming pattern? We believe these findings can help guide tool builders and researchers in novel, fruitful directions that formalize and improve naming practices among developers.

11 Declarations

11.1 Funding

This work was not funded by any agency

11.2 Ethical approval

This work does not involve human or animal subjects and does not require IRB approval

11.3 Informed consent

This work does not involve human or animal subjects and does not require informed consent

11.4 Author Contributions

1. Conceptualization: Christian D. Newman
2. Data curation: Christian D. Newman, Anthony Peruma, Syreen Banabilah, Michael J. Decker, Reem S. AlSuhaibani, Eman Abdullah Alomar, Mahie Crabbe
3. Formal analysis: Christian D. Newman
4. Investigation: Christian D. Newman
5. Methodology: Christian D. Newman
6. Project administration: Christian D. Newman
7. Resources: Christian D. Newman
8. Software: Christian D. Newman, Farhad Akhbardeh, Anthony Peruma
9. Supervision: Christian D. Newman
10. Validation: Christian D. Newman, Anthony Peruma, Syreen Banabilah, Eman Abdullah Alomar
11. Visualization: Christian D. Newman
12. Writing – original draft: Christian D. Newman, Jonathan I Maletic, Mohamed Wiem Mkaouer, Marcos Zampieri
13. Writing – review & editing: Christian D. Newman, Jonathan I Maletic, Anthony Peruma, Syreen Banabilah, Michael J. Decker, Reem S. AlSuhaibani, Eman Abdullah Alomar, Marcos Zampieri

11.5 Data Availability Statement

We have created a repository that contains the data and scripts needed to generate the numbers and statistical analysis from the RQs. The scripts are in the scripts directory and need only to be run (they take no arguments). The data directory contains all annotation data broken down by closed category. Each file is named

after the category it contains, and they are amalgamated in a single file (called Tagger Open Coding). This repository can be found at this link[5].

11.6 Conflict of Interests

We have no competing/conflicting interests to report

11.7 Clinical Trial Number

Not Applicable

[5] https://github.com/SCANL/closed_category_emse_analysis_scripts

References

1. Abebe, S.L., Tonella, P.: Towards the extraction of domain concepts from the identifiers. In: Proceedings of the 2011 18th Working Conference on Reverse Engineering, WCRE '11, p. 77–86. IEEE Computer Society, USA (2011). DOI 10.1109/WCRE.2011.19. URL https://doi.org/10.1109/WCRE.2011.19
2. Abebe, S.L., Tonella, P.: Automated identifier completion and replacement. In: 2013 17th European Conference on Software Maintenance and Reengineering, pp. 263–272 (2013). DOI 10.1109/CSMR.2013.35
3. Adolph, S., Hall, W., Kruchten, P.: Using grounded theory to study the experience of software development. Empirical Software Engineering **16**, 487–513 (2011). DOI 10.1007/s10664-010-9152-6
4. Al Madi, N.: Namesake: A checker of lexical similarity in identifier names. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, ASE '22. Association for Computing Machinery, New York, NY, USA (2023). DOI 10.1145/3551349.3560441. URL https://doi.org/10.1145/3551349.3560441
5. Allamanis, M., Barr, E.T., Bird, C., Sutton, C.: Suggesting accurate method and class names. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, pp. 38–49. ACM, New York, NY, USA (2015). DOI 10.1145/2786805.2786849. URL http://doi.acm.org/10.1145/2786805.2786849
6. Alsuhaibani, R., Newman, C., Decker, M., Collard, M., Maletic, J.: On the naming of methods: A survey of professional developers. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp. 587–599 (2021). DOI 10.1109/ICSE43902.2021.00061
7. Alsuhaibani, R.S., Newman, C.D., Collard, M.L., Maletic, J.I.: Heuristic-based part-of-speech tagging of source code identifiers and comments. In: 2015 IEEE 5th Workshop on Mining Unstructured Data (MUD), pp. 1–6 (2015). DOI 10.1109/MUD.2015.7327960
8. Aman, H., Amasaki, S., Yokogawa, T., Kawahara, M.: A quantitative investigation of trends in confusing variable pairs through commits: Do confusing variable pairs survive? In: Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering, EASE '24, p. 90–99. Association for Computing Machinery, New York, NY, USA (2024). DOI 10.1145/3661167.3661228. URL https://doi.org/10.1145/3661167.3661228
9. Arnaoudova, V., Di Penta, M., Antoniol, G., Guéhéneuc, Y.: A new family of software anti-patterns: Linguistic anti-patterns. In: 2013 17th European Conference on Software Maintenance and Reengineering, pp. 187–196 (2013). DOI 10.1109/CSMR.2013.28
10. Arnaoudova, V., Eshkevari, L.M., Penta, M.D., Oliveto, R., Antoniol, G., Gueheneuc, Y.G.: Repent: Analyzing the nature of identifier renamings. IEEE Trans. Softw. Eng. **40**(5), 502–532 (2014). DOI 10.1109/TSE.2014.2312942. URL https://doi.org/10.1109/TSE.2014.2312942
11. Avidan, E., Feitelson, D.G.: Effects of variable names on comprehension: An empirical study. In: 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC), pp. 55–65 (2017). DOI 10.1109/ICPC.2017.27
12. Binkley, D., Hearn, M., Lawrie, D.: Improving identifier informativeness using part of speech information. In: Proceedings of the 8th Working Conference on Mining Software Repositories, MSR '11, pp. 203–206. ACM, New York, NY, USA (2011). DOI 10.1145/1985441.1985471. URL http://doi.acm.org/10.1145/1985441.1985471
13. Binkley, D., Lawrie, D., Morrell, C.: The need for software specific natural language techniques. Empirical Softw. Engg. **23**(4), 2398–2425 (2018). DOI 10.1007/s10664-017-9566-5. URL https://doi.org/10.1007/s10664-017-9566-5
14. Butler, S., Wermelinger, M., Yu, Y.: A survey of the forms of java reference names. In: 2015 IEEE 23rd International Conference on Program Comprehension, pp. 196–206 (2015). DOI 10.1109/ICPC.2015.30
15. Butler, S., Wermelinger, M., Yu, Y., Sharp, H.: Relating identifier naming flaws and code quality: An empirical study. In: 2009 16th Working Conference on Reverse Engineering, pp. 31–35 (2009). DOI 10.1109/WCRE.2009.50
16. Butler, S., Wermelinger, M., Yu, Y., Sharp, H.: Exploring the influence of identifier names on code quality: An empirical study. In: Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on, pp. 156–165. IEEE (2010)
17. Butler, S., Wermelinger, M., Yu, Y., Sharp, H.: Mining java class naming conventions. In: 2011 27th IEEE International Conference on Software Maintenance (ICSM), pp. 93–102 (2011). DOI 10.1109/ICSM.2011.6080776

18. Caprile, Tonella: Restructuring program identifier names. In: Proceedings 2000 International Conference on Software Maintenance, pp. 97–107 (2000). DOI 10.1109/ICSM.2000.883022
19. Caprile, C., Tonella, P.: Nomen est omen: analyzing the language of function identifiers. In: Sixth Working Conference on Reverse Engineering (Cat. No.PR00303), pp. 112–122 (1999). DOI 10.1109/WCRE.1999.806952
20. Collard, M.L., Maletic, J.I.: srcml 1.0: Explore, analyze, and manipulate source code. In: 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 649–649 (2016). DOI 10.1109/ICSME.2016.36
21. Corbi, T.A.: Program understanding: Challenge for the 1990s. IBM Systems Journal **28**(2), 294–306 (1989). DOI 10.1147/sj.282.0294
22. Corbin, J., Strauss, A.: Grounded theory research: Procedures, canons and evaluative criteria. Qualitative Sociology **19**(6), 3–21 (1990). DOI https://doi.org/10.1007/BF00988593. URL `https://doi.org/10.1515/zfsoz-1990-0602`
23. Deissenboeck, F., Pizka, M.: Concise and consistent naming. Software Quality Journal **14**(3), 261–282 (2006). DOI 10.1007/s11219-006-9219-1. URL `https://doi.org/10.1007/s11219-006-9219-1`
24. Deissenboeck, F., Ratiu, D.: 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)
25. Dragan, N., Collard, M.L., Maletic, J.I.: Reverse engineering method stereotypes. In: Proceedings of the 22Nd IEEE International Conference on Software Maintenance, ICSM '06, pp. 24–34. IEEE Computer Society, Washington, DC, USA (2006). DOI 10.1109/ICSM.2006.54. URL `http://dx.doi.org/10.1109/ICSM.2006.54`
26. Etgar, A., Friedman, R., Haiman, S., Perez, D., Feitelson, D.G.: The effect of information content and length on name recollection. In: Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, ICPC '22, p. 141–151. Association for Computing Machinery, New York, NY, USA (2022). DOI 10.1145/3524610.3529159. URL `https://doi.org/10.1145/3524610.3529159`
27. Fakhoury, S., Roy, D., Ma, Y., Arnaoudova, V., Adesope, O.: Measuring the impact of lexical and structural inconsistencies on developers' cognitive load during bug localization. Empirical Softw. Engg. **25**(3), 2140–2178 (2020). DOI 10.1007/s10664-019-09751-4. URL `https://doi.org/10.1007/s10664-019-09751-4`
28. Falleri, J.R., Huchard, M., Lafourcade, M., Nebut, C., Prince, V., Dao, M.: Automatic extraction of a wordnet-like identifier network from software. In: Proceedings of the 2010 IEEE 18th International Conference on Program Comprehension, ICPC '10, p. 4–13. IEEE Computer Society, USA (2010). DOI 10.1109/ICPC.2010.12. URL `https://doi.org/10.1109/ICPC.2010.12`
29. Fry, Z.P., Shepherd, D., Hill, E., Pollock, L., Vijay-Shanker, K.: Analysing source code: looking for useful verb-direct object pairs in all the right places. IET Software **2**(1), 27–36 (2008). DOI 10.1049/iet-sen:20070112
30. Glassman, E.L., Fischer, L., Scott, J., Miller, R.: Foobaz: Variable name feedback for student code at scale. Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology (2015). URL `https://api.semanticscholar.org/CorpusID:15810023`
31. Gupta, S., Malik, S., Pollock, L., Vijay-Shanker, K.: Part-of-speech tagging of program identifiers for improved text-based software engineering tools. In: 2013 21st International Conference on Program Comprehension (ICPC), pp. 3–12 (2013). DOI 10.1109/ICPC.2013.6613828
32. Hellendoorn, V.J., Bird, C., Barr, E.T., Allamanis, M.: 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, p. 152–162. Association for Computing Machinery, New York, NY, USA (2018). DOI 10.1145/3236024.3236051. URL `https://doi.org/10.1145/3236024.3236051`
33. Hill, E.: Integrating natural language and program structure information to improve software search and exploration. Ph.D. thesis, Newark, DE, USA (2010). AAI3423409
34. Hill, E., Fry, Z.P., Boyd, H., Sridhara, G., Novikova, Y., Pollock, L., Vijay-Shanker, K.: 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, p. 79–88. Association for Computing Machinery, New York, NY, USA (2008). DOI 10.1145/1370750.1370771. URL `https://doi.org/10.1145/1370750.1370771`

35. Hoda, R.: Socio-technical grounded theory for software engineering. IEEE Transactions on Software Engineering **48**(10), 3808–3832 (2022). DOI 10.1109/TSE.2021.3106280
36. Hofmeister, J., Siegmund, J., Holt, D.V.: Shorter identifier names take longer to comprehend. In: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 217–227 (2017). DOI 10.1109/SANER.2017.7884623
37. Host, E., Ostvold, B.: The programmer's lexicon, volume i: The verbs. pp. 193 – 202 (2007). DOI 10.1109/SCAM.2007.18
38. Høst, E.W.: Meaningful method names (2011)
39. Høst, E.W., Østvold, B.M.: Debugging method names. In: Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming, Genoa, pp. 294–317. Springer-Verlag, Berlin, Heidelberg (2009). DOI 10.1007/978-3-642-03013-0_14. URL `http://dx.doi.org/10.1007/978-3-642-03013-0_14`
40. Høst, E.W., Østvold, B.M.: The java programmer's phrase book. In: D. Gašević, R. Lämmel, E. Van Wyk (eds.) Software Language Engineering, pp. 322–341. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
41. Høst, E.W., Østvold, B.M.: Canonical method names for java. In: B. Malloy, S. Staab, M. van den Brand (eds.) Software Language Engineering, pp. 226–245. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
42. Hucka, M.: Spiral: splitters for identifiers in source code files. Journal of Open Source Software **3**, 653 (2018). DOI 10.21105/joss.00653
43. Kashiwabara, Y., Onizuka, Y., Ishio, T., Hayase, Y., Yamamoto, T., Inoue, K.: 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), pp. 323–327 (2014). DOI 10.1109/CSMR-WCRE.2014.6747186
44. Lawrie, D., Morrell, C., Feild, H., Binkley, D.: What's in a name? a study of identifiers. In: 14th IEEE International Conference on Program Comprehension (ICPC'06), pp. 3–12 (2006). DOI 10.1109/ICPC.2006.51
45. Liblit, B., Begel, A., Sweetser, E.: Cognitive perspectives on the role of naming in computer programs. In: In Proc. of the 18th Annual Psychology of Programming Workshop (2006)
46. Liu, H., Liu, Q., Liu, Y., Wang, Z.: Identifying renaming opportunities by expanding conducted rename refactorings. IEEE Transactions on Software Engineering **41**(9), 887–900 (2015)
47. Liu, H., Liu, Q., Staicu, C.A., Pradel, M., Luo, Y.: Nomen est omen: Exploring and exploiting similarities between argument and parameter names. In: Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on, pp. 1063–1073. IEEE (2016)
48. Liu, K., Kim, D., F. Bissyandé, T., Kim, T., Kim, K., Koyuncu, A., Kim, S., Le Traon, Y.: 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)
49. Malik, R.S., Patra, J., Pradel, M.: Nl2type: Inferring javascript function types from natural language information. In: Proceedings of the 41st International Conference on Software Engineering, ICSE '19, p. 304–315. IEEE Press (2019). DOI 10.1109/ICSE.2019.00045. URL `https://doi.org/10.1109/ICSE.2019.00045`
50. Martin, R.C.: Clean Code: A Handbook of Agile Software Craftsmanship, 1 edn. Prentice Hall PTR, Upper Saddle River, NJ, USA (2008)
51. Miller, G.A.: Wordnet: a lexical database for english. Communications of the ACM **38**(11), 39–41 (1995)
52. Munaiah, N., Kroh, S., Cabrey, C., Nagappan, M.: Curating github for engineered software projects. Empirical Software Engineering **22**(6), 3219–3253 (2017). DOI 10.1007/s10664-017-9512-6. URL `https://doi.org/10.1007/s10664-017-9512-6`
53. Newman, C., Decker, M., Alsuhaibani, R.: Identifier name structure catalogue URL `https://github.com/SCANL/identifier_name_structure_catalogue.` [Online]. Available: https://github.com/SCANL/identifier_name_structure_catalogue.
54. Newman, C.D., AlSuhaibani, R.S., Collard, M.L., Maletic, J.I.: Lexical categories for source code identifiers. In: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 228–239 (2017). DOI 10.1109/SANER.2017.7884624
55. Newman, C.D., AlSuhaibani, R.S., Decker, M.J., Peruma, A., Kaushik, D., Mkaouer, M.W., Hill, E.: On the generation, structure, and semantics of grammar patterns in source code identifiers. Journal of Systems and Software **170**, 110740 (2020). DOI https://doi.org/10.1016/j.jss.2020.110740. URL `https://www.sciencedirect.com/science/article/pii/S0164121220301680`

56. Newman, C.D., Decker, M.J., AlSuhaibani, R.S., Peruma, A., Kaushik, D., Hill, E.: An empirical study of abbreviations and expansions in software artifacts. In: Proceedings of the 35th IEEE International Conference on Software Maintenance. IEEE (2019)
57. Newman, C.D., Decker, M.J., Alsuhaibani, R.S., Peruma, A., Mkaouer, M.W., Mohapatra, S., Vishnoi, T., Zampieri, M., Sheldon, T.J., Hill, E.: An ensemble approach for annotating source code identifiers with part-of-speech tags. IEEE Transactions on Software Engineering **48**(9), 3506–3522 (2022). DOI 10.1109/TSE.2021.3098242
58. Olney, W., Hill, E., Thurber, C., Lemma, B.: Part of speech tagging java method names. In: 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 483–487 (2016). DOI 10.1109/ICSME.2016.80
59. Osumi, Y., Umekawa, N., Komata, H., Hayashi, S.: Empirical study of co-renamed identifiers. In: 2022 29th Asia-Pacific Software Engineering Conference (APSEC), pp. 71–80 (2022). DOI 10.1109/APSEC57359.2022.00019
60. Parsa, S., Zakeri-Nasrabadi, M., Ekhtiarzadeh, M., Ramezani, M.: Method name recommendation based on source code metrics. Journal of Computer Languages **74**, 101177 (2023). DOI https://doi.org/10.1016/j.cola.2022.101177. URL `https://www.sciencedirect.com/science/article/pii/S2590118422000740`
61. Peruma, A., Hu, E., Chen, J., AlOmar, E.A., Mkaouer, M.W., Newman, C.D.: Using grammar patterns to interpret test method name evolution. In: 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC), pp. 335–346 (2021). DOI 10.1109/ICPC52881.2021.00039
62. Peruma, A., Mkaouer, M.W., Decker, M.J., Newman, C.D.: 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`
63. Peruma, A., Mkaouer, M.W., Decker, M.J., Newman, C.D.: 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)
64. Peruma, A., Mkaouer, M.W., Decker, M.J., Newman, C.D.: Contextualizing rename decisions using refactorings, commit messages, and data types. Journal of Systems and Software **169**, 110704 (2020). DOI https://doi.org/10.1016/j.jss.2020.110704. URL `http://www.sciencedirect.com/science/article/pii/S0164121220301503`
65. Peruma, A., Newman, C.D.: Understanding digits in identifier names: An exploratory study. In: Proceedings of the 1st International Workshop on Natural Language-Based Software Engineering, NLBSE '22, p. 9–16. Association for Computing Machinery, New York, NY, USA (2023). DOI 10.1145/3528588.3528657. URL `https://doi.org/10.1145/3528588.3528657`
66. Ratiu, D., Deissenboeck, F.: Programs are knowledge bases. pp. 79 – 83 (2006). DOI 10.1109/ICPC.2006.41
67. Ratiu, D., Deissenboeck, F.: From reality to programs and (not quite) back again. In: Proceedings of the 15th IEEE International Conference on Program Comprehension, ICPC '07, p. 91–102. IEEE Computer Society, USA (2007). DOI 10.1109/ICPC.2007.22. URL `https://doi.org/10.1109/ICPC.2007.22`
68. Schankin, A., Berger, A., Holt, D.V., Hofmeister, J.C., Riedel, T., Beigl, M.: Descriptive compound identifier names improve source code comprehension. In: Proceedings of the 26th Conference on Program Comprehension, ICPC '18, pp. 31–40. ACM, New York, NY, USA (2018). DOI 10.1145/3196321.3196332. URL `http://doi.acm.org/10.1145/3196321.3196332`
69. Shepherd, D., Fry, Z.P., Hill, E., Pollock, L., Vijay-Shanker, K.: 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, pp. 212–224. ACM, New York, NY, USA (2007). DOI 10.1145/1218563.1218587. URL `http://doi.acm.org/10.1145/1218563.1218587`
70. Singer, J., Kirkham, C.: Exploiting the correspondence between micro patterns and class names. In: 2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation, pp. 67–76 (2008). DOI 10.1109/SCAM.2008.23
71. Stol, K.J., Ralph, P., Fitzgerald, B.: Grounded theory in software engineering research: A critical review and guidelines (2016). DOI 10.1145/2884781.2884833
72. Takang, A.A., Grubb, P.A., Macredie, R.D.: The effects of comments and identifier names on program comprehensibility: an experimental investigation. J. Prog. Lang. **4**, 143–167 (1996)

73. Toutanova, K., Manning, C.D.: 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, pp. 63–70. Association for Computational Linguistics, Stroudsburg, PA, USA (2000). DOI 10.3115/1117794.1117802. URL https://doi.org/10.3115/1117794.1117802
74. van der Werf, V., Swidan, A., Hermans, F., Specht, M., Aivaloglou, E.: Teachers' beliefs and practices on the naming of variables in introductory python programming courses. In: Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training, ICSE-SEET '24, p. 368–379. Association for Computing Machinery, New York, NY, USA (2024). DOI 10.1145/3639474.3640069. URL https://doi.org/10.1145/3639474.3640069
75. Zhang, J., Liu, S., Gong, L., Zhang, H., Huang, Z., Jiang, H.: Beqain: An effective and efficient identifier normalization approach with bert and the question answering system. IEEE Transactions on Software Engineering **49**(4), 2597–2620 (2023). DOI 10.1109/TSE.2022.3227559