Supporting the Maintenance of Identifier Names: A Holistic Approach to High-Quality Automated Identifier Naming

Anthony S. Peruma
axp6201@rit.edu

Supporting the Maintenance of Identifier Names: A Holistic Approach to High-Quality Automated Identifier Naming

by

Anthony S. Peruma

A dissertation submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
in Computing and Information Sciences

B. Thomas Golisano College of Computing and
Information Sciences

Rochester Institute of Technology
Rochester, New York
June 2022

Supporting the Maintenance of Identifier Names:
A Holistic Approach to High-Quality Automated Identifier Naming

by

Anthony S. Peruma

Committee Approval:

We, the undersigned committee members, certify that we have advised and/or supervised the candidate on the work described in this dissertation. We further certify that we have reviewed the dissertation manuscript and approve it in partial fulfillment of the requirements of the degree of Doctor of Philosophy in Computing and Information Sciences.

Dr. Christian D. Newman Dissertation Advisor	Date
Dr. Mohamed W. Mkaouer Dissertation Committee Member	Date
Dr. Marcos Zampieri Dissertation Committee Member	Date
Dr. Mehdi Mirakhorli Dissertation Committee Member	Date
Dr. Robert Glick Dissertation Defense Chairperson	Date

Certified by:

Dr. Pengcheng Shi Ph.D. Program Director, Computing and Information Sciences	Date

Supporting the Maintenance of Identifier Names: A Holistic Approach to High-Quality Automated Identifier Naming

by

Anthony S. Peruma

Submitted to the
B. Thomas Golisano College of Computing and Information Sciences Ph.D. Program in Computing and Information Sciences
in partial fulfillment of the requirements for the
Doctor of Philosophy Degree
at the Rochester Institute of Technology

Abstract

A considerable part of the source code is identifier names– unique lexical tokens that provide information about entities, and entity interactions, within the code. Identifier names provide human-readable descriptions of classes, functions, variables, etc. Poor or ambiguous identifier names (i.e., names that do not correctly describe the code behavior they are associated with) will lead developers to spend more time working towards understanding the code's behavior. Bad naming can also have detrimental effects on tools that rely on natural language clues; degrading the quality of their output and making them unreliable. Additionally, misinterpretations of the code, caused by poor names, can result in the injection of quality issues into the system under maintenance. Thus, improved identifier naming increases developer effectiveness, higher-quality software, and higher-quality software analysis tools.

In this dissertation, I establish several novel concepts that help measure and improve the quality of identifiers. The output of this dissertation work is a set of identifier name appraisal and quality tools that integrate into the developer workflow. Through a sequence of empirical studies, I have formulated a series of heuristics and linguistic patterns to evaluate the quality of identifier names in the code and provide naming structure recommendations. I envision and working towards supporting developers in integrating my contributions, discussed in this dissertation, into their development workflow to significantly improve the process of crafting and maintaining high-quality identifier names in the source code.

Acknowledgments

Accomplishing a work of this magnitude would not be possible single-handedly. I am eternally grateful to the many individuals who supported, advised, and encouraged me.

My sincerest thanks to my advisor, Dr. Christian D. Newman, for his constant dedication and guidance. His vast knowledge and experience not only gave me a solid foundation to conduct my research, but also helped me grow as a research scientist. I am also greatly indebted to Dr. Mohamed W. Mkaouer for his advice on my research and for navigating me through the complex maze of academia. I am also thankful to all my collaborators I worked with throughout the years for giving me the chance to build my research portfolio. I would also like to acknowledge the undergraduate software engineering senior project teams I had the privilege of working with and their contributions to my research. I also convey my thanks to my committee members for their insightful comments and suggestions.

I am also ever grateful to the staff and faculty members of the Department of Computing and Information Sciences and the Department of Software Engineering for supporting me during my academic journey.

I will be failing in my duties if I forget to acknowledge my friends, both near and far, who provided me with a constant source of encouragement throughout my Ph.D. program. Last but not least, my family– thank you for all your encouragement and support that empowered me to pursue and achieve my Ph.D. in Computing and Information Sciences.

To my family, for all of their endless love, support, and encouragement.

∞

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Regardless of industry or technical domain, quality is a critical aspect of any software system [134]. Furthermore, ensuring the quality of a software system is not a one-time task; starting from implementation and continuing throughout the system's lifetime, developers perform maintenance tasks on their software to meet functional and non-functional goals [100]. This emphasis on software quality has resulted in organizations dedicating between 60% to 80% of resources to software maintenance [158, 198], thus being the costliest phase of the software development lifecycle.

A key element of software maintenance is program comprehension [200]. Program comprehension is the act of developers reading source code to either understand the purpose of the code or to identify the statements related to their maintenance activity. Prior to making changes to the code that facilitate the evolution of the system, developers need to read the lines of code in the source files to understand the behavior of the code [162, 220]. Naturally, it is not surprising that issues such as poor code readability and understandability not only impact the time developers take to perform their tasks but can also impact the quality of the updates performed on the system. Furthermore, differences in the background (e.g., skill, experience) between the original author of the code and the maintainer also impacts comprehension [199]. With 58% of a developer's time spent on comprehension activities [223], it is essential that developers craft source code such that it does not hinder readability and understandability. This includes making improvements to the code ranging from design level changes, such as reducing cyclomatic complexity [197] to the compliance of consistency, in the form of naming conventions [210].

As fundamental elements in the source code, identifier names account for almost 70% of the characters in a software system's codebase [124]. These names are lexical tokens that uniquely identify entities in the code (such as classes, methods, variables, etc.) and play a significant part in code comprehension. Indeed, the importance of good identifier names is recognized by both industry and academia. This importance is reflected through software engineering practices, which provide guidelines [136, 167], best practices [135, 164], metrics, and models [112, 205] to assist developers in naming identifiers with the overall goal of improving code comprehension [116]. Therefore, developers must pay significant attention when constructing an identifier's name as their choices impact the time spent understanding the identifier's purpose [155, 206]; as well-constructed names can improve comprehension activities by as much as 19% [144].

1.1 Problem Statement

While quality metrics, best practices, and guidelines highlight the need for high-quality identifier names, they only act as heuristics for developers; they are not formal mechanisms to produce strong names. These mechanisms cannot help developers use the correct wording, nor can they recommend the lexical structure beyond what is heuristically correct (e.g., naming conventions). For instance, the example presented in Listing 1.1 is a clear indication of a poor identifier name that can be detected using standard naming conventions. However, determining the quality of the identifier names in Listing 1.2 and 1.3 is not straightforward. In these two examples, the names follow a specific naming style (i.e., camel case), utilize a known set of terms, and are readable. However, they do not accurately reflect the intended behavior of the identifier. In Listing 1.2, the identifier's name is singular, but its associated data type is a collection; does this identifier represent a collection of URLs or a single URL? In Listing 1.3, the method name suggests a transformation of a given object, but the method does not return a value. Furthermore, quality metrics around readability are generally unable to capture readability improvements [181].

```java
@Override
protected boolean func_22246_a(int var1)
{
    return false;
}
```

Listing 1.1: An example of a poor quality identifier name (`'func_22246_a'` [169]).

```java
private HashSet includeCrawlingURL;
```

Listing 1.2: An example of a poor but readable identifier name (`'includeCrawlingURL'` [98]).

To correct poor identifier names, like those shown in the above examples, developers perform a rename refactoring operation [133] on the identifier's name, which either preserves the original

```java
public void toSAX(ContentHandler ch) throws SAXException {
    XMLByteStreamInterpreter deserializer = new XMLByteStreamInterpreter()
        ;
    deserializer.setContentHandler(new EmbeddedXMLPipe(ch));
    deserializer.deserialize(this.xmlBytes);
}
```

Listing 1.3: An example of a poor but readable identifier name (‘`toSAX`’ [99]).

```java
@Override
protected boolean isSelected(int var1)
{
    return false;
}
```

Listing 1.4: An example of a renamed identifier name (‘`func_22246_a`’ → ‘`isSelected`’ [169]).

meaning of the name or changes it [101]. For example, to correct the poor quality identifier name in Listing 1.1, the developer performs a *Rename Method* refactoring operation, the results of which yield a high-quality name, as shown in Listing 1.4. To this extent, most current approaches, proposed by the research community, rely on detecting and suggesting identifier rename opportunities in the source code, using machine learning techniques [88, 89, 159]. However, as these approaches are dependent on the existing code, pre-existing poor identifier names in the source code will negatively influence the generated name's quality; they may even result in the injection of quality issues into the system under maintenance. Thus, their fully automated, big-data-driven nature may work against them. A stronger understanding of naming structures from the code-behavior and linguistic-semantics perspectives can help us correctly train these tools to not only leverage historical data but also pay attention to a human-derived understanding of name meaning based on both linguistic and program-oriented information. In other words, we can use human-derived empirical data to direct these tools to better solutions than if we only rely on algorithmically derived patterns.

My work leverages manually, partially-automated data which is then used to train and direct fully-automated techniques to understand the meaningful structures and semantics behind identifier names and their evolution. My studies leading up to my dissertation examine the evolution of names in the source code to identify linguistic patterns around the semantic and part-of-speech changes a name undergoes alongside the co-occurrence of renames with changes to the surrounding code. These findings take the form of rules that are incorporated into productivity tools to check for naming violations/anti-patterns and assist developers with not only crafting identifier names but also helping them better manage the identifiers in the source code of their projects.

1.2 Goal

In my work, I study the quality of identifier names with the goal of *supporting developers in appraising, structuring/improving, and maintaining identifier names.* The tangible outcome of my Ph.D. program is a framework of tools that are constructed based on empirical, scientific data that, when integrated into the developer workflow, evaluates the quality of existing identifier names and provides high-quality name and naming structure recommendations. The framework consists of an IDE plugin and a command-line tool. The IDE plugin provides developers with real-time recommendations for which word type (e.g., verb, noun) and structure developers should utilize for high-quality name replacements. The offline tool, which supports integration into a build system, reports on linguistic anti-patterns (i.e., deviations from well-established lexical naming practices) in the source code.

1.3 Research Questions

To this extent, my dissertation answers the following research questions:

- **RQ$_1$: How effectively, in terms of correctness, can we generate identifier name structure recommendations?** This RQ examines the relationships between an identifier's grammar pattern and its surrounding code to determine the extent to which the relationship can assist with appraising and recommending the semantic structure of the name.
- **RQ$_2$: To what extent does the proposed automated recommendation approach, based on the semantic structure of a name, positively or negatively influence naming practices?** This RQ involves evaluating an automated identifier name structure appraisal and recommendation technique to understand its effectiveness and overall usability.
- **RQ$_3$: What are the primary challenges in appraising and recommending the semantic structure of identifier names, and how can these be improved?** This RQ performs an error analysis to identify and understand the weaknesses in the proposed name structure appraisal and recommendation approach and determine a course of action to address these challenges.

1.4 Research Focus Areas

In this section, I elaborate on the research areas that I have focused on to achieve my end goal. In brief, there are three areas– (1) identifier name evolution, (2) tool development, and (3) developer workflow integration.

1.4.1 Identifier Name Evolution

This area involves the study of how names evolve over time. This includes analyzing how words are changed (e.g., hyponym, synonym relationships), how code changes influence these name changes, and how development activities also trigger name changes. This analysis provides me with historical data about identifier name evolution and the causes/reasoning behind this evolution, which will be essential for constructing a tool that needs to integrate into a developer's workflow and provide well-motivated advice.

1.4.2 Tool Development

This research area involves incrementally (i.e., as needed) improving basic technologies used to analyze and transform words. This includes improving the accuracy of part-of-speech taggers on source code. Many existing natural language processing technologies are not built to work on source code, or are still inaccurate even when they work. As these are challenging problems to solve, improvements will be made incrementally. Thus, it allows me to make progress in studying how to provide helpful feedback to developers, which will become better as basic technologies improve.

1.4.3 Developer Workflow Integration

This area involves integrating my tools into the developer workflow. The suggestions made by my IDE plugin and console application need to integrate into a developer's workflow smoothly. It cannot be distracting and should not provide the developer with useless suggestions. Thus, it is essential to study the effectiveness of the tools in a developer setting.

1.5 Contribution

The contributions from my dissertation range from multiple large-scale empirical studies that examine developer naming practices to tools that assist developers with constructing and maintaining high-quality identifier names in the source code. More specifically, the findings from my empirical studies lead to the formation of heuristics incorporated into tools, which are in the form of an IDE plugin and console application that can be integrated with a build, continuous integration, or code review system.

Furthermore, through my research studies, I have released datasets of identifier names and name evolution in open-source systems and a catalog of linguistic anti-patterns. I envision that these artifacts will spur more research in the field and support improved rigor in the renaming process.

Finally, my completed studies have been published in peer-reviewed, high-quality software engineering research venues (refer to Appendix A for the complete listing of publications).

1.6 Organization

This dissertation is organized as follows:

- Chapter 2 provides a background around the lexical-semantic concepts utilized in the studies within this dissertation.
- Chapter 3 presents related work around identifier names, including studies that examine the quality of an identifier's name and rename suggestions.
- Chapters 4 to 11 provide more in-depth details about the completed studies in this area. The work performed in these studies feeds into answering the RQs of my dissertation. These studies include empirical studies and tool development papers. To be specific:
  - Chapters 4,5 6, 7 and 8 are the empirical studies I conducted on large and diverse datasets.
  - Chapters 9 and 10 provide details on the identifier naming tools I have implemented.
  - Chapters 11 are details of a user study conducted on an IDE plugin for appraising and recommending the semantic structure of identifier names in the code.
- Chapter 12 provides a discussion on how my completed studies answer my proposed research questions.
- Finally, in Chapter 13, I summarize my dissertation and provide insight into my future work in this area.

Chapter 2

Lexicosemantics

Lexicosemantics (or lexico-semantics or lexical semantics) is a branch of linguistics that is concerned with the study of word meanings, such as the internal semantic structure of words and the relationship between the different senses of a word [122]. In this chapter, I discuss the different concepts utilized in my studies to determine the quality of an identifier's name.

2.1 Taxonomy for Rename Refactorings

The research studies in this dissertation use a rename-specific taxonomy originally established by Arnaoudova et al. [101] to assess rename refactorings and group them into different kinds based on their semantics. The taxonomy is briefly discussed in this section.

Entity Kind: Entity kind records the source code entity that a given identifier represents. For example, the identifier may be the name of a type, class, getter, setter, etc.

Form of Renaming: The identifier's lexical change is reflected in this category and consists of four subcategories: Simple, Complex, Reordering, and Formatting. Simple identifier changes are ones that just add, remove, or replace one term. Multiple terms are added, removed, or changed in Complex alterations. Reordering occurs when two or more terms in an identifier change positions (for example, NameEmployee becomes EmployeeName), while formatting changes occur when a letter in a term changes case or a separator (for example, an underscore) is added or removed.

Semantic Changes: These are modifications to the meaning of the identifier as a result of adding/removing terms or modifying terms (e.g., using a term that is a synonym or antonym of the original). To determine if the semantics of an identifier have been preserved or modified, the following heuristics are utilized.

An identifier's meaning is preserved if one of the following holds: 1) the change added/removed a separator, 2) the change expanded an abbreviation, 3) the change collapsed a term into an abbreviation, 4) the old term was changed to a new term which is a synonym of the old term, 5) multiple old terms were changed to multiple new terms which are synonyms OR use or removal of negation preserves the meaning of the identifier (i.e., ItemNotVisible becomes ItemHidden).

An identifier's meaning is modified if one of the following holds: 1) *Broaden meaning*– the old term is renamed to a hypernym of itself OR a term (i.e., adjective or noun) was removed which generalizes the identifier (e.g., GetEmployeeFirstName becomes GetEmployeeName). 2) *Narrowing meaning*– the old term is renamed to a hyponym of itself OR a term was removed which narrows the meaning of the identifier (e.g., GetEmployeeName becomes GetEmployeeFirstName). 3) *Meaning changed* (i.e., not narrowed or broadened)– when an old term is changed to a new term that is unrelated to the old; when a new term is the old term's meronym/holonym, or antonym; OR when multiple terms are changed AND a negation reverses a synonym of the old term. 4) *Add meaning*– one or more new terms were added to the identifier AND the addition does not fall into one of the categories above (e.g., narrow meaning). 5) *Remove meaning*– one or more terms removed from the identifier AND the removal does not fall into one of the categories above (e.g., broaden meaning).

2.1.1 Contextualizing Rename Refactorings

Identifiers are renamed by developers for a variety of reasons. One can acquire insight into how developers chose their words, why they prefer some sorts of words over others, and how to automate this process by carefully analyzing rename refactorings. This subsection shows examples of how developer activity, recorded in commit messages and refactoring operations, is reflected in their renaming choices.

By analyzing the following method rename: *setDisableBinLogCache* → *setEnableReplicationCache*, it is observed that the meaning of the name has changed; the developer has modified the name by changing *disable* to *enable*. This change is reflected in the commit message entered by the developer: *"Changes replication caching to be disabled by default"* [20]. Similarly, the renaming of a class from *Key* → *EntityKey* demonstrates an act of narrowing the meaning of the identifier. Once again, the purpose of this rename is reflected in the commit message: *"Rename Key to EntityKey to prepare specialized caches"* [26].

Developers may also rename identifiers to: 1) better represent the existing functionality and not when they are changing or narrowing it, or 2) adhere to naming standards or correct a spelling/-grammatical mistake. For example, here the developer renamed the class *TestProxyController*

→ *ProxyControllerTest* by reordering the term names to *"...fixed names that were not in standards"* [15]. In the next example, the developer preserves the meaning of a method by renaming it from *inactivate* → *deactivate*, through the use of a synonym. This is, again, reflected in the commit message: *"Renaming method to proper English..."* [9], where renaming to 'proper English' indicates that the meaning has not been modified but should now be easier to comprehend.

Finally, commit messages are not the only way to contextualize rename refactorings. Changes to the code surrounding a name also help in understanding the developer's intention. Unfortunately, most types of changes to the code are not part of a pre-defined taxonomy. That is, it is difficult to understand the abstract, domain-level goal of individual changes. Luckily, some types of code changes are taxonomized. Specifically, refactorings are a taxonomy of changes made to the code for a specific goal; typically to optimize non-functional attributes of the code [133]. We can look at refactorings that happen just before and right after a given rename to help us understand what the developer was doing before and after they applied a rename refactoring.

For example, in commit [30] the developers applied an *Extract Method* refactoring with the following comment: "using the Jangaroo parsing infrastructure; all tests green; getters inherited", before applying rename: *getCompilationsUnit* → *getCompilationUnit*. This preserves the meaning of the name but puts the name more in-line with its type, as stated by the commit message for this change: "Corrected type in internal method name" [31].

Another example comes from a move class refactoring, where a class was moved from one package to another [48]. This refactoring commit had the following comment: "Incremental changes, some package refactorings etc". Further, a rename was performed after this commit: *JsonViewResult*→*JsonView* [47]. This rename broadens the meaning of the name by removing *result*, making the identifier more general in meaning. The commit message associated with the rename is: "Cleaned up some file names for easier usage...", meaning the developer was likely going through and renaming things after the move class refactoring.

In addition to surrounding code changes, a change in the data type associated with an identifier can also help contextualize a rename of an identifier. For example, in commit [53], the developer performs the following *Rename Variable*: *Date sqlDate* → *Timestamp timestamp* with the commit message "fixes issue #29. java.util.Date and jodatime.Datetime instances would loose time information..." From this example, we see that reason for the rename is to fix a bug by utilizing the `Timestamp` data structure instead of `Date`.

Table 2.1: Examples of grammar patterns

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

2.2 Grammar Patterns

To understand the relationship between groups of identifiers, I use grammar patterns in my research. The sequence of part-of-speech tags (also known as annotations) assigned to individual words within an identifier's name is known as a *grammar pattern*. For example, an identifier called GetUserToken is assigned a grammar pattern by splitting the identifier into its three constituent words: Get, User, and Token. The split-sequence (Get Employee Name) is then passed through a part-of-speech tagger to determine the grammar pattern: Verb Noun Noun. This grammar pattern is not unique to this identifier; it is shared by many other potential identifiers who utilize similar terms. Thus, a grammar pattern can be used to connect identifiers that contain different terms; GetEmployeeName, SetUserId, and WriteEmployeeAddress all have the same grammar pattern, and while they do not express the same semantics, their grammatical patterns indicate similarities in their semantics. Specifically, a verb (get, set, write) that is applied to a noun (name, id, address) that has a specified role or context (employee, user). Table 2.1 shows a set of identifiers on the left and the corresponding grammar pattern on the right.

2.2.1 Noun, Verb, and Prepositional phrases

A few linguistic concepts emerge when studying the output of a part-of-speech tagger, specifically when dealing with noun phrases, verb phrases, and prepositional phrases. A Noun Phrase (NP) is a sequence of noun modifiers, such as noun-adjuncts and adjectives, followed by a noun, and optionally followed by other modifiers or prepositional phrases [163]. The noun in a noun phrase is typically referred to as a *head-noun*; the entity which is being modified/described by the words to its left [125] (or, for programmers, sometimes surrounding it) in the phrase. A Verb Phrase (VP) is a verb followed by an NP and optionally a Prepositional Phrase (PP). A PP is a preposition plus

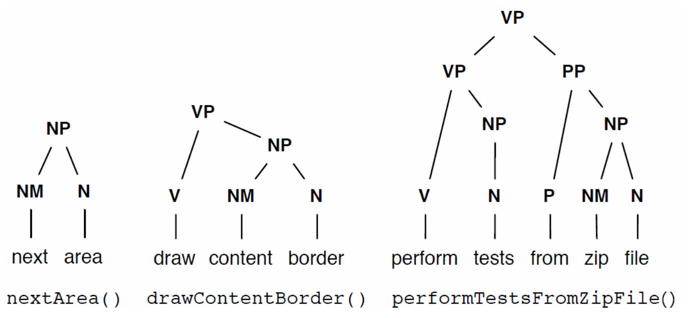


Figure 2.1: Examples of noun, verb, and prepositional phrases

an NP and can be part of a VP or NP.

Figure 2.1 presents an example NP, VP, and VP with PP for three method name identifiers. The phrase structure nodes are NP, VP, and PP, while the other nodes (i.e., N, NM, V, P) are part-of-speech annotations. The leaf nodes are the individual words split from within the identifier. Each word in the identifier is assigned a part-of-speech, which can then be used to derive the identifier's phrase structure. A phrase structure cannot be built without part-of-speech information. One important thing to note about these phrases is how the words in the phrases work together. For example, in noun phrases, noun modifiers (e.g., other nouns or adjectives) work to modify (i.e., specify) the concept represented by the head-noun that is part of the same phrase. In Figure 2.1, *contentBorder* is a noun phrase where *content* modifies the understanding of the noun *border*. This example shows that border refers to a content border as opposed to another type of border; a *window border*, for example. When converted into a verb phrase by adding draw to get *drawContentBorder*; an action is added (i.e., draw) that will be applied to the particular type of border (i.e., the content border) represented by the identifier.

2.3 Linguistic Anti-Patterns

Linguistic anti-patterns are deviations from well-established lexical naming conventions in source code that serve as indicators of poor name quality [103]. This loss of quality causes inconsistencies in the source code, which leads to misinterpretations and an increase in developer cognitive effort [129]. Detecting naming violations in source code is a time-consuming and error-prone operation for developers that necessitates a thorough grasp of the system as well as a manual analysis of the entire source code. As a result, tool support is required.

Chapter 3

Related Work

Since the choice of adequate naming for identifiers is critical for code understandability, there have been many studies that analyze the quality of identifiers and how identifier quality affects comprehension and developer efficiency. In this chapter, I divide my reporting of related work into three areas– studies that explore the naming of test methods, studies that investigate identifier quality attributes, grammar patterns in identifier names, and studies around the renaming of identifiers in source code.

3.1 Identifier Name Quality

There are several recent approaches to appraising identifier names for variables, methods, and classes. Liu et al. [161] propose an automated approach based on deep learning to debug method names based on consistency between the method's name and its implementation. Kashiwabara et al. [152] use association rule mining to identify verbs that might be good candidates for use in method names; this work focuses on word co-occurrence to find any emergent relationships. Abebe and Tonella [84] use an ontology that models the word relationships within a piece of software. They then generate suggestions for new identifier names using different schemes for how to choose sequences of words to put together to form the identifier.

Liblit et al. [157] discuss naming in several programming languages and make observations about how natural language influences the use of words in these languages. Schankin et al. [206] focus on investigating the impact of more informative identifiers on code comprehension. Their findings show an advantage of descriptive identifiers over non-descriptive ones. Hofmeister et al. [144] compare comprehension of identifiers containing words against identifiers containing letters and/or abbreviations. Their results show that identifier names containing only words instead of abbreviations or letters, increase developer comprehension speed by 19% on average. Lawrie et al. [155] study 100+ programmers, asking them to describe twelve different functions. These functions use three different levels of identifiers: single letters, abbreviations, and full words. The results show that full word identifiers lead to the best comprehension, though there were cases where there was no statistical difference between full words and abbreviations. Butler et al. [118] extend their previous work on java class identifiers [114] to show that method identifiers with flaws are also (i.e., along with class identifiers) associated with low-quality code according to static analysis-based metrics.

Høst and Østvold [145] design automated naming rules using method signature elements, i.e., return type, parameter names, and types, and control flow. They call this technique method phrase refinement, which takes a sequence of part-of-speech tags (i.e., phrases) and concretizes them by substituting real words. (e.g., the phrase <verb>-<adjective> might refine to is-empty). Additionally, they use static analysis to group method names (in phrase form) together by behavior.

Arnaoudova et al. [102] define a catalog of linguistic anti-patterns that are found to deteriorate the quality of code understanding. The authors show the negative impact of linguistic anti-patterns by conducting two studies with software developers and finding that the majority of programmers perceive anti-patterns as poor naming practices. In their study of readability metrics, Fakhoury et al. [130] show that current metrics may not be effective at capturing readability improvements; highlighting the importance of further research into the quality of naming and how names evolve over time.

3.2 Identifier Grammar Patterns

In an empirical study on 5,000 open-source projects, Zhang et al. [227] observe that nouns, verbs, and adjectives are three of the most common part-of-speech tags developers utilize in crafting identifier names. The authors utilize Standford Parser to parse the part-of-speech tags from an identifier's name automatically. Binkley et al. [106] investigate the effectiveness of the Stanford log-linear part-of-speech tagger on field names. Through this study, the authors propose four rules, based on part-of-speech tags, for improving field names. A study of naming in multiple programming languages by Liblit et al. [157] shows how natural language influences the use of words in these languages. Høst and Østvold [145] examine unusual method names and propose a set of naming rules to uncover issues in method names. The authors utilize part-of-speech tags along with the return type, control flow, and parameters of the method to detect naming violations based on a set of rules.

3.3 Identifier Renaming

Allamanis et al. [88] introduce a model, NATURALIZE, that utilizes statistical natural language processing to mine and learn the style (i.e., coding conventions) of a codebase and provides renaming suggestions. NATURALIZE learns syntactic restrictions, or sub-grammars, on identifier names like camelcase or underscore and to unify names used in similar contexts. The authors also propose a neural probabilistic language model to suggest descriptive, idiomatic method and class names automatically [89]. Suzuki et al. [212] introduce an n-gram based model for evaluating the comprehensibility of method names and suggesting comprehensible method names. In their approach, the authors collect and learn method names from open-source Java systems. As part of their analysis strategy, the authors utilize a threshold to determine the comprehensibility score of a method's name and use the n-gram model to make suggestions to the developer. Research by Liu et al. [159] looks at recommending renames based on the prior rename activities developers perform on the source code. Additionally, by studying the relationship between argument and parameter names, the authors develop an approach to detect naming anomalies and suggest renames to developers [160]. In their study, Jiang et al. [147] observe that the effectiveness of *code2vec*, a machine learning-based approach for method name recommendations, fails in a realistic setting. The authors also propose a heuristic-based approach that outperforms *code2vec*. Liu et al. [161] utilize deep learning techniques to identify inconsistent method names. Their approach extracts extract deep representations of method names and bodies. The authors train the model using a large number of methods from real-world projects. The name suggestion approach involves a comparison of overlap between the closeness of method names in the method name vector space and the set of methods names whose bodies are close in the method body vector space. Arnoudova et al. [101] 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.

Research Focus Area: Identifier Name Evolution

Chapter 4

An Empirical Investigation of How and Why Developers Rename Identifiers

The contents of this chapter are part of the study "***An Empirical Investigation of How and Why Developers Rename Identifiers***" published in the Proceedings of the 2$^{\text{nd}}$ International Workshop on Refactoring [191].

4.1 Introduction

The majority of software life-cycle resources are allocated to program maintenance [110,128]. Maintenance heavily relies on program comprehension since developers typically spend a significant portion of their time in understanding the code they are maintaining before applying changes, debugging, documenting, etc. It is clear that making it easier to comprehend code will ease many maintenance activities and improve developer productivity. One of the primary ways for a developer to come to terms with what a body of code is doing is through the identifiers in the code. It has been stated that identifiers make up an estimated 70% of characters within a software system [125]. Because of this, their meaningful naming is critical to the program's comprehension. When an identifier's name no longer appropriately describes the role of the identifier in the software system, a developer will change the name. This change is called the rename refactoring. Rename refactorings are a common type of refactoring and are part of Fowler's taxonomy [133]. Rename refactorings modify non-functional attributes of a software system (i.e., the name of the identifier). In renaming an identifier, the new name should be better suited to describe the identifier's role in the current state of the system than the old name. The study of rename refactorings is gaining more attention in research; it is well understood that we need a stronger understanding of how natural language is used to support comprehension and how it evolves with the software.

Many techniques to support comprehension rely on the analysis of identifiers [138, 141, 211, 226]. Furthermore, many previous studies have investigated naming practices, patterns, and how to improve analysis of identifier names [103, 106, 143, 154, 175]. In particular, a number of papers explore the idea of debugging, appraising, and generating identifier names [84, 89, 145]. These have a direct, positive impact on approaches that synthesize programs [97, 166, 203], which must understand how developers describe code elements (e.g., name identifiers, comment on methods) in order to generate natural language that developers will accept (i.e., text that optimizes developer comprehension). Thus, there is a need in the research community for analyzing identifier naming practices, especially when performed by developers in real-world scenarios.

In this study, we present an analysis on the evolution of method, class, and package identifiers using rename refactorings over the history of 3,795 Java systems and a total of 524,113 identifiers. The goal of this study is to extend a portion of the work done by Arnaoudova et al. [101] to a much larger number of systems and combine analysis based on their taxonomy with commit messages to investigate why developers rename identifiers in particular ways. We aim to begin understanding why, for example, a developer chooses to narrow or generalize the meaning of an identifier. To do so, we address 5 research questions. Similar to Arnaoudova et al. [101], the first three research questions are primarily defined to explore our findings on rename practices. The last two questions analyze the rename refactorings we have collected to classify the developer's rename practices using the context of their development efforts as well as to provide preliminary results for our future research.

RQ1: What Types of Lexical Changes Are Typically Applied By a Rename Refactoring? What proportion of renames are just a change to a single or to multiple terms in an identifier, what proportion are changes to the order of terms, changes in plurality, changes in capitalization, or addition of separators. For this research question, we want to know at a high-level what renames look like. This will give us an idea of how complex renames tend to be and give us some measure of how much (in terms of constituent words) identifiers tend to change.

RQ2: What Kinds of Semantic Changes Occur to Terms Composing Identifiers When They Are Renamed? What types of changes to an identifier's meaning are most frequent? The goal of this question will be to explore how often an identifier's meaning is broadened, narrowed, preserved, completely changed, added to, or removed from. The answer to this question will help us determine typical renaming behavior and help provide finer-grain insight into rename activity.

RQ3: What Kinds of Grammar Changes Occur to Terms Composing Identifiers When They Are Renamed? We want to know when there is a change in the part of speech tag for any individual term in an identifier. The answer to this question helps complete our view of what changed. A change in part of speech does not always mean the term completely changed; some part of speech changes are due to a change in the original term's conjugation. Most importantly, we want to know if it can help us determine when a semantic change has occurred.

RQ4: To What Extent Can Commit Messages Be Used to Contextualize Different Types of Semantic Change Rename Refactorings? If we use topic modeling on a corpus of commit messages grouped by semantic change category, can we begin reasoning about what types of activities are undertaken by developers when they make different types of semantic changes?

RQ5: What Trends Do We See in the Way Identifiers Are Renamed? Finally, taking all of the data we have gathered, can we identify any development activities that correlate with different types of semantic changes made to identifiers? This question uses the data we gathered and uses it to help us understand what causes different types of semantic changes made to identifiers during software evolution.

The results help gain an understanding of the causes and consequences of rename refactorings. In particular, we see the results as the first step towards better supporting tools that try to understand developer behavior when it comes to naming. This will eventually help increase the adoption of technology that supports identifier name evolution and the creation of better guidelines for how renames should be applied and supported during maintenance.

4.2 Methodology

We conduct a two-phased approach to answer our research questions. The initial phase consists of the retrieval of open-source Java projects and the detection of refactoring operations that occur throughout the development history of each retrieved project. The second phase of the experiment involves the analysis of the detected renaming operations as a means of understanding the type of approaches utilized by developers when changing identifier names. Figure 4.1 depicts the flow of steps involved in this experiment. Described below are details of each phase.

4.2.1 Data Collection & Refactoring Detection

To ensure that our study accurately captures real-world identifier renaming operations it was imperative that our research be based on a representative dataset. To this extent, our study utilizes the list of GitHub based Java projects made available by [90]. To identify refactoring operations performed by developers on these projects, we use RefactoringMiner [208] on each project. By

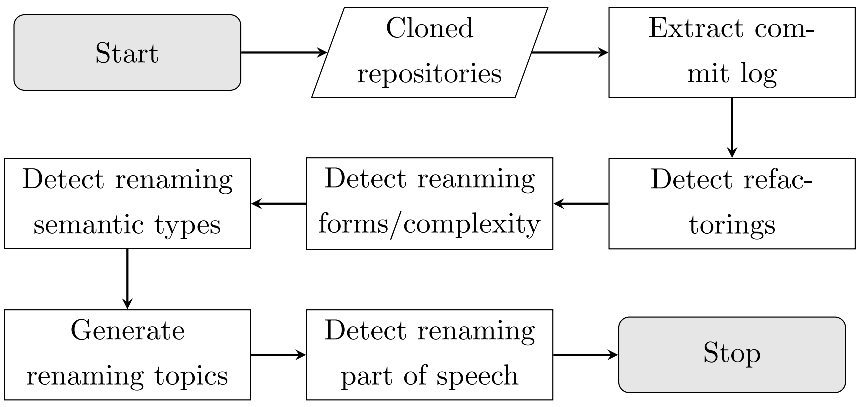

Start
Cloned repositories
Extract commit log
Detect refactorings
Detect reanming forms/complexity
Detect renaming semantic types
Generate renaming topics
Detect renaming part of speech
Stop

Figure 4.1: Overview of Experiment Methodology

enumerating through the commit history of each project, RefactoringMiner is able to detect over 1,000,000 refactoring operations in 3,795 projects. As shown in Table 4.1, from the detected refactoring types, 43.36% refactoring operations are related to renaming operations (i.e., Package, Class and Method renaming). Furthermore, developers tend to perform more rename operations on method names when compared to class or package names; with projects, on average, containing approximately 9 package, 47 class, and 122 method renames. Hence, not surprisingly, the number of commits associated with method renames is also significantly higher. However, the occurrences of the different types of rename refactorings in the projects, contained in our dataset, have similar distributions.

4.2.2 Rename Analysis

To understand the renaming changes made by developers, we perform a tool-based taxonomy analysis on the original and renamed identifier names. Since we were unable to obtain a copy of REPENT, we attempt to stick as close to the tools and technologies they report using in the original study [101]. To this end, we utilize the Natural Language Toolkit (NLTK, `https://www.nltk.org/`), which has an implementation of Wordnet [168], to obtain semantic and part of speech details about the identifier names.

Prior to performing our analysis, we perform preprocessing on the original and new names of the renamed identifiers. Given that most identifier are composed of multiple terms, our approach involved splitting each name into a list of terms (i.e., tokenization). To perform the splitting, we utilize the Ronin splitter algorithm implemented in the Spiral package [146]. Table 4.2 provides an overview of the most frequent number of terms that constitute the name of the identifier for

Table 4.1: Detected Refactorings - Data Overview

Refactoring Operation	Count	Percentage
Refactoring operation occurrences		
Rename Package	18,372	1.52%
Rename Class	129,206	10.69%
Rename Method	376,535	31.15%
Others	684,857	56.65%
Projects containing refactoring operations		
Rename Package	1,875	16.32%
Rename Class	2,735	23.80%
Rename Method	3,086	26.86%
Others	3,795	33.03%
Commits containing refactoring operations		
Rename Package	12,516	2.44%
Rename Class	54,590	10.66%
Rename Method	122,600	23.94%
Others	322,479	62.96%

Table 4.2: Most Frequent Number of Terms Forming an Identifier Name

Identifier type	Terms in orig. name	Terms in new name	Total occ	***Percentage***
Package	1	1	12,672	***68.97%***
Class	3	3	17,387	***13.46%***
Method	3	3	66,080	***17.55%***

packages, classes, and methods.

Our semantic analysis follows the approach presented in [101]. We compare each term of the original and new identifier for semantic relationships such as synonyms, hyponym, hypernym, antonym, meronyms, and holonyms. If a relationship does not exist, we perform a stem-based check between the two identifiers and re-compare them. We utilize the Porter, Lancaster, and Snowball stemming algorithms for this purpose. We also perform a lemmatization check using NLTK's algorithm. We used multiple stemming/lemmatization techniques to try and find as many matches as possible. For every detected match, each of which we call a *matched term*, we also derive the part of speech associated with the new and original terms. Additionally, we use their heuristics [101] to determine semantic changes and fully described in the original work. Specifically, we identify renames that preserve, change, narrow, broaden, adds, or removes meaning from the old to the new identifier. We also detect the complexity of the rename (# terms changed), term reordering, formatting changes, and addition/removal of terms.

4.3 Experimental Results

4.3.1 RQ1: What Types of Lexical Changes Are Typically Applied By a Rename Refactoring?

As shown in Table 4.3, the majority of renamings fall under the Simple category, meaning developers change only one term via rename. Furthermore, a breakdown of Simple renamings shows that approximately 57% of the renamings involved replacing a single term in the name (e.g., core → engine) while 24% involve the addition of a term (e.g., QueueFactory → TaskQueueFactory) and 19% involve the removal of a term (e.g., RewriteEventBase → RewriteBase). Complex renamings are the next most frequent categorization at 35% of detected renamings. We observe that the two most common types of complex renaming patterns are the addition of two terms along with the removal of a single term (e.g., RecentURLEvent → RecentResourceNamesEvent) and the

Table 4.3: Forms of Identifier Renamings

Form	Package	Class	Method	***Total***	**%**
Formatting	210	3,271	24,236	***27,717***	***5.29%***
Reordering	3	2,479	2,223	***4,705***	***0.90%***
Simple	14,072	77,722	21,5651	***307,445***	***58.66%***
Complex	4,087	45,734	134,425	***184,246***	***35.15%***
Total	***18,372***	***129,206***	***376,535***	***524,113***	***100.00%***
Percentage	***3.51%***	***24.65%***	***71.84%***	***100.00%***	

removal of two terms along with the addition of a single term (e.g., FormOpenIdLoginServlet → FormAuthLoginServlet) occurring at 18% and 17% respectively in the dataset. Formatting and reordering are the least frequent, indicating that most renames are not just changes to the format (e.g., adding separator or change capitalization). All in all, while simple renamings are the most common, complex renamings are very common as well; both should be high priority for study.

4.3.2 RQ2: What Kinds of Semantic Changes Occur to Terms Composing Identifiers When They Are Renamed?

Table 4.4 contains the distribution of different categories of semantic changes made by rename refactorings. We observe from this table that developers most frequently narrow the meaning of identifiers (44.8%) when performing a rename. One reason for this may be that developers initially construct identifier names that reflect a generalized or incomplete understanding of the ultimate functionality, and through updates they specialize the name of the identifier to reflect increasing understanding or specialization of the entity (i.e., class, method). Another reason may be that the system evolves and functionality specializes as a part of this evolution. Conversely, renamings that broadened the meaning of the word accounted for approximately 11% of matches; significantly less than narrow meaning.

Looking at the preserved meaning category, approximately 42% of matches in this category are synonym based. For example, when the developer renames the class from *DefaultLocationProvider* → *DefaultLocationSupplier*, the developer replaces the term *Provider* with a synonym, *Supplier*. Another 36% of the term matches are only after stemming (comprising of 20%, 16%, and 0.07% Porter, Lancaster and Snowball matches respectively). For example, renaming a method name from *checkInitialize* → *checkInitialized* results in a stem match for the terms *Initialize* and *Initialized* using the Porter stemming algorithm. Moreover, the last 22% of the term matches are detected

Table 4.4: Frequency of Semantic Change Rename Refactorings

Category	Package	Class	Method	***Total***	%
Preserve	1,091	9,659	30,054	***40,804***	***7.8%***
Change	4	112	1,140	***1,256***	***0.2%***
Narrow	744	77,606	156,804	***235,154***	***44.9%***
Broaden	467	16,489	41,283	***58,239***	***11.1%***
Add	310	11,910	74,657	***86,877***	***16.6%***
Remove	131	2,045	12,158	***14,334***	***2.7%***
None	15,625	11,385	60,439	***87,449***	***16.7%***
Total	***18,372***	***129,206***	***376,535***	***524,113***	***100%***
Percentage	***3.51%***	***24.65%***	***71.84%***	***100%***	

only after using the NLTK WordNet Lemmatizer. Conversely, we find remarkably few instances where a rename changes the meaning of an identifier (0.24%) according to the heuristics we use.

Finally, we also detect occurrences of identifier renamings that either added or removed a meaning to the identifier name. For example, a developer renaming of a method from *targetNode* → *getTargetNode* adds the term *get* to the new name to better describe the purpose of the method. Similarly, renaming a method name from *applyTo* → *apply* removes the term *To* from the new name. Add meaning is more common (16.6%) than remove meaning (2.73%). This lends support to the idea that identifiers generally narrow in meaning over time.

4.3.3 RQ3: What Kinds of Grammar Changes Occur to Terms Composing Identifiers When They Are Renamed?

As depicted in Figure 4.2, the majority of matched terms in the renamed identifiers demonstrate grammar (i.e., part of speech) changes. For example, a part of speech change occurs when renaming the class from *ProfilingDataSource* → *ProfiledDataSource*. The term *Profiling* in the original name is a present participle verb. This term is replaced with a past participle verb, *Profiled*, in the new name.

The most common grammar changes are depicted in Table 4.6, where we see that verb → noun, noun (singular) → noun (plural), and verb (present) → verb (past) make up the bulk of all grammar changes. Given this, an important follow up question is whether these actually indicate a change or modification of the identifier's meaning with respect to the heuristics we use. While changing

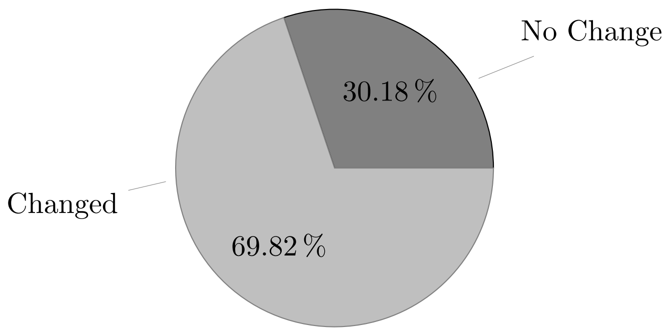


Figure 4.2: Part of Speech Changes for 42,060 Matched Terms

Table 4.5: Total Number of Grammar Changes Detected Correlated with Semantic Change

Semantic change type	***Total***	***Percentage***
Grammar change preserved meaning	***29,110***	***99.13%***
Grammar change changed meaning	***256***	***0.87%***

from singular → plural or past → present are not likely candidates, changing from verb → noun may correlate with some modification in meaning.

We investigated this question, and while verb → noun changes are very common, there is very little evidence that the grammar change is indicative of a change or modification in the meaning of the identifier. In fact, there is a significant amount of evidence that any grammar change indicates preservation of the identifier's meaning after a rename. As reported in Table 4.5, the majority of the part of speech changes fall under the preserve meaning semantic category. A breakdown on the preserve meaning-based changes shows that synonym-based matches contributed to most of the part of the speech changes, closely followed by lemmatized matches; an overview is provided in Table 4.6. This suggests that the grammar changes are primarily caused when a term in an identifier is changed to a closely related term (i.e., synonym) or a different inflection of the term (i.e., lemma or stem).

4.3.4 RQ4: To What Extent Can Commit Messages Be Used to Contextualize Different Types of Semantic Change Rename Refactorings?

To begin understanding more about why identifier meanings change in different ways (e.g., narrow in meaning), we use commit message text associated with a refactoring based rename. Our approach involves the use of Latent Dirichlet Allocation (LDA) [109] to discover and generate topics contained within the commit messages automatically. To this end, we utilize the LDA imple-

Table 4.6: Most Frequent Preserve-Meaning Part of Speech Changes

Preserve meaning match type	Original term part of speech	New term part of speech	***Total***	***Percentage***
Synonym	Verb	Noun (singular)	***13,178***	***45.27%***
WordNet Lemmatizer	Noun (singular)	Noun (plural)	***8,752***	***30.07%***
Porter Stemmer	Verb (present)	Verb (past)	***5,226***	***17.95%***
Lancaster Stemmer	Verb (past)	Noun (singular)	***1,927***	***6.62%***
Snowball Stemmer	Noun (singular)	Adverb	***27***	***0.09%***

mentation contained within gensim [201], a Python-based topic modeling library. To ensure the generation of reliable topics, the commit messages are first pre-processed before the generation of the LDA model. The pre-processing task includes the removal of stopwords (via NLTK's corpus of stopwords), removal of numeric values and stemming of individual words. We then agglomerate all commit messages into seven corpora; one for each of the semantic change categories (i.e., preserve, change, narrow, broaden, etc) and run LDA on each corpus independently.

We show an overview of the topics generated by LDA in Table 4.7 for each of the seven types of semantic change classifications described in this study including the *none* category. Next to each word in parenthesis is the probability score assigned by LDA. While we generate five words and five topics for each semantic change category, Table 4.7 only presents the top 2 topics (labeled A and B) for each category. Lastly, it is worth noting that for each topic we present the number of words processed to show how many words LDA uses to form these topics. Almost every category has more than 100k words with the exception of Remove meaning and Change meaning. Change meaning, in particular, containes very few words and the quality of the output from LDA suffers as a result.

The first observation we note in Table 4.7 is that the words *test*, *rename*, and *fix* have a relatively higher score according to LDA. This indicates that developers perform renames in test- or fix-related commits very often. In the case of *fix*, it seems likely that the renames occurr after a developer addressed some issue and felt that identifier names need to be modified in light of the changed code. It also seems relatively frequent for developers to explicitly state that they are performing a rename within the commit message, since the term *rename* appears as the most relevant word frequently.

Examining the rest of the table, there is more variance in the words that appear. Words like *ad* and *add* (where *ad* is a stemmed version of *add*) appear relatively frequently in the narrow, broaden, and add meaning categories. This may indicate that adding code correlates with these types of renames. Because narrow meaning is the most common semantic change type (Table 4.4), it could be that as code evolves and grows, identifier names tend to narrow in meaning overall and, failing that, they become longer. That is, they specialize as code specializes. This does not hold in all cases, seeing as how add and ad occur with a broaden meaning, which makes up 58k (11%) of semantic change renamings we detected. Change meaning has too few words in its corpora compared to the other categories. The results from LDA are weaker relative to other categories and the terms that appear in its topics are not exclusively unique to it.

Preserve and Remove meaning lack *ad* or *add* in their commit messages at the topic level, which is not surprising. If we assume adding tends to modify meaning somehow (i.e., narrow, broaden, add) then Preserve and Remove should not include these terms. Instead, terms like *rename* and *refactor* are more common in these and Add meaning than in others. Interestingly, Remove meaning does not include terms like *remove*, *delete*, etc. It is also the category with the second least number of terms LDA has to work with after Change meaning. Thus, like Change meaning, results here are less likely to generalize well compared to some of the other categories. The only other term in Remove meaning that is relatively significant compared to other topics is *method.*

Finally, the None category contains a number of terms that are linked with package renamings. This may indicate that the heuristics we use from the taxonomy underperform on package renamings. Regardless, package renamings are more likely to end up in the None category. In this regard, this preliminary observation is subject of investigation in our future work in order to correctly categorize package renamings.

To summarize, there are interesting trends in the way identifiers are renamed and the types of activities developers are undertaking according to commit messages. While it is difficult to pinpoint the developer's intention by only analyzing at the level of code, our findings do provide avenues for future research.

4.3.5 RQ5: What Trends Do We See in the Way Identifiers Are Renamed?

In RQ1, using Table 4.3, we find that most changes are simple; they only change one term (59% are simple). However, a significant number of them change multiple terms (35% are complex). This is to say that a large number of renames modify two or more terms, which is the majority of terms with respect to the most frequent size of identifiers in method and classes (Table 4.2). Additionally, from RQ2, using Table 4.4, we know that a significant portion of these narrow, broaden, or add (45%, 17%, and 11% respectively) to the meaning of the identifier. To get a better handle on this data, we turn to Table 4.8, which shows how many simple and complex renamings are categorized

Table 4.7: LDA Generated Topics for Semantic-Based Renamings

Topic	Word #1	Word #2	Word #3	Word #4
Classification Type: ***Preserve***			*Words Processed: 101,378*	
A	renam (0.073)	method (0.023)	clean (0.013)	refactor (0.012)
B	test (0.067)	fix (0.060)	refactor (0.027)	issu (0.019)
Classification Type: ***Change***			*Words Processed: 3,274*	
A	test (0.027)	fix (0.015)	ad (0.014)	chang (0.013)
B	name (0.022)	chang 0.019)	fix (0.016)	class (0.014)
Classification Type: ***Narrow***			*Words Processed: 473,541*	
A	test (0.081)	fix (0.030)	ad (0.022)	add (0.014)
B	fix (0.049)	ad (0.029)	add (0.026)	support (0.024)
Classification Type: ***Broaden***			*Words Processed: 99,163*	
A	test (0.086)	ad (0.032)	fix (0.023)	chang (0.016)
B	add (0.034)	support (0.022)	ad (0.015)	api (0.013)
Classification Type: ***Add***			*Words Processed: 141,678*	
A	test (0.090)	ad (0.034)	fix (0.023)	class (0.012)
B	fix (0.086)	renam (0.046)	issu (0.034)	method (0.029)
Classification Type: ***Remove***			*Words Processed: 30,373*	
A	test (0.082)	chang (0.017)	fix (0.015)	name (0.015)
B	renam (0.029)	method (0.029)	refactor (0.025)	chang (0.016)
Classification Type: ***None***			*Words Processed: 187,878*	
A	test (0.066)	fix (0.062)	name (0.026)	packag (0.019)
B	renam (0.049)	java (0.017)	core (0.016)	org (0.013)

Table 4.8: Renaming-Complexity for Semantic Changes

Category	Simple	Complex	***Total***	***Percentage***
Preserve	18,577	21,853	***40,430***	***8.22%***
Change	616	612	***1,228***	***0.25%***
Narrow	146,391	86,512	***232,903***	***47.37%***
Broaden	48,934	9,297	***58,231***	***11.84%***
Add	58,428	28,082	***86,510***	***17.59%***
Remove	9,459	4,875	***14,334***	***2.92%***
None	25,040	33,015	***58,055***	***11.81%***
Total	***307,445***	***184,246***	***491,691***	***100.00%***
Percentage	***62.53%***	***37.47%***	***100.00%***	

under each category of semantic changes. The table shows that the majority of both complex and simple renamings narrow the meaning of their identifiers, with Add meaning and Broaden meaning being the next two most common. We also know from RQ4 that commit messages most frequently reference adding, fixing, tests in reference to these three categories.

We can surmise from this that future work should take a closer look at all categories, but these three in particular (at least, at the level of methods, classes, and packages). While analyzing the commit message gives us a high-level view of why different types of renames are applied in practice, a more fine grain analysis is required to understand what is going on at the level of source code. We can use data from these commit message trends to begin exploring the relationship between, for example, adding code and narrowing the meaning of identifiers. Additionally, in RQ3 we observe that grammar changes are strongly correlated with preservation of identifier meaning. While more study is required, this result is interesting and can be used to model grammar changes that indicate preservation versus those which require further investigation.

The conclusion we draw for this research question is that rename refactorings (whether simple or complex) clearly narrow the meaning of identifiers more often than not. More research is required to fully understand why and what types of changes or activities preface different types of semantic changes, but having identified the most prolific categories and gained some high-level understanding of what developers report when performing these changes, we can now focus on these activities to help us understand how changes to (and particularly addition of) code affect the decision to rename and how to differentiate between source code level additions that cause, for example, a narrowing of identifier meaning.

Chapter 5

Contextualizing Rename Decisions using Refactorings, Commit Messages, and Data Types

The contents of this chapter are part of the study "***Contextualizing rename decisions using refactorings, commit messages, and data types***" published in the Journal of Systems and Software [192].

5.1 Introduction

Program comprehension is the pillar of a developer's everyday development tasks; almost every programming task requires a certain degree of understanding of the existing codebase. In this way, software maintenance and evolution critically rely on the degree to which developers comprehend their codebases. Many studies demonstrate the significance of the effort, in terms of time, undertaken by developers when comprehending code [121, 164]. For instance, the time spent by developers in reading and comprehending code is significantly longer than the time spent in writing new instructions [164]. Therefore, developers' productivity can be optimized by decreasing the time needed for them to understand the existing code [144, 155, 206, 213].

One of the most atomic activities in source code development is the naming of code elements (e.g., class names, function/method names, etc.) collectively referred to as identifiers. Identifier names are the basic building blocks of program comprehension. Choices made when constructing identifier names directly impacts productivity [118,144,155,206]. For example, abbreviated terms may hinder comprehension for both tools and humans. In response, studies search for ways to standardize and normalize identifier names to support both developers and tools [107, 176] and many research projects, both recent and otherwise, aim to enhance identifier naming using machine learning, static analysis, and by studying naming inconsistencies [84, 89, 145, 152, 161, 179].

One way to improve identifiers is to apply a rename refactoring [133]. Rename refactorings are defined as refactorings that modify the name of an identifier without modifying the intended behavior of the code for which the identifier is a part. Many Integrated Developer Environments (IDEs) offer a built-in rename refactoring functionality. Most of these IDEs only support the mechanical act of renaming; they allow a developer to choose what identifier they want to rename, what new name should be used, and then perform checks to ensure that the new name will not introduce name collisions and that the new name is applied in all appropriate locations. There is little or no support to help inform developers of when to rename an identifier (e.g., when a name is of sub-optimal quality), and how to rename them (i.e., what words to use within the name). Instead, renames are typically performed when a developer notices that an identifier does not accurately reflect the behavior it represents. This causes renaming to be applied in a manner that is not always wholly systematic. Further, a developer is free to come up with whatever name they like (i.e., within the limits of naming conventions defined for the project). This new name may be even worse than the original, but there is no formal method to determine when this is the case.

Because naming heavily affects comprehension for both tools and humans, it is important to fully support developers when they must modify identifier names. That is, research must support developers in applying rename refactorings. Recent research on naming focuses heavily on suggesting identifier names [84,89,152,161], studying how names correlate with behavior [103,145], and analyzing names to reveal interesting properties [94,101,106,137,175]. We focus this work on investigating how names evolve [101, 159, 160, 185, 191] (i.e., are changed via rename) and how these changes affect/are affected by: 1) other changes made to the code (i.e., behavior preserving or not) as part of 2) a larger development plan/context. This information is critical if we are to support the evolution of identifier names by recommending when and how to rename an identifier.

In this study, we study renames in two ways. 1) This study utilizes a taxonomy of rename types published by Arnoudova et al. [101] to understand the types of changes applied to identifier names within our dataset. That is, we study how individual terms within an identifier are modified both syntactically and semantically when a rename refactoring is applied. 2) The study contextualizes these rename types by analyzing changes to data types, commit log data, and refactorings that co-occur with renames. This allows us to understand how changes, to the code surrounding an identifier, affect changes to the identifier's name and, likewise, how development activities (i.e., written in a commit log) affect changes to the identifier's name.

This study is an extension of two prior works. Our initial work on renames investigated how method, class, and package identifier names evolve and how this evolution was described in commit messages. Our aim was to understand how names evolve and how this evolution is documented. Our assumption was that the choice of wording used during a rename is influenced by external factors which may appear in commit messages [191]. Through this work, we determined some preliminary trends in how development activities, such as adding features, influenced the terminology used during a rename. However, due to the limitations of natural language analysis techniques, and the occasionally information-light nature of commit messages, the results we found were not as actionable as we wanted. The trends we detected were nevertheless instructive and helped guide us toward ways to improve our approach. Thus, we extended [191] by taking into account the types of refactorings which occur before, or after, a rename refactoring while performing commit message analysis on these surrounding operations [185]. This allowed us to more clearly identify how names are influenced by their surrounding changes (e.g., *Extract Method* and *Move Attribute* frequently occur before certain types of renames) and how these influences are documented in commit messages. It also highlighted further challenges, which we discussed as research directions for ourselves and the larger software engineering research community.

In this study, we extend our recent work [185] by additionally considering the situation where a rename is applied to an identifier, and that identifier's corresponding data type is changed. Data type changes are interesting because, unlike refactorings, they may change the external behavior of their associated identifier. Data types tell us what data and behavior an identifier represents (i.e., the data type tells us what attributes and methods can be used with this identifier). Therefore, when an identifier and its data type both change, this indicates a potential shift in behavior (e.g., added methods, new API), a shift in the data represented by an identifier (e.g., added attribute), or shift in the representation of data and behavior in a system (e.g., a change to improve comprehension). By studying this situation, we can gain a more acute understanding of name evolution using a type of code change (i.e., data type changes) that has a stronger, direct influence on the behavior of a given identifier, and provide means for previous rename recommendation techniques to consider type migration as another dimension to learn a more suitable name.

The goal of this extension is to understand the influence that data type changes have on the structure and meaning of a rename. We emphasize data type changes for three reasons: 1) Changes to an identifier's type are relatively easy to detect in many programming languages. Therefore, making suggestions to developers on the fly when a type change is performed is already feasible in modern IDEs. 2) Types have a strong influence over the data and behavior represented by an identifier, so changes to the type can have heavy significance on their associated identifier. 3) Type changes are a simple way for us to explore some non-refactoring code changes related to renames. These results will provide insight into our long-term goals. In the long term, the outcomes of this study will be used to support research into: 1) recommending when a rename should be applied (e.g., after specific types of refactorings), 2) recommending how to rename an identifier (e.g., what words to use), and 3) developing a model that describes how developers mentally synergize names using domain and project knowledge. We provide insights into how our data can support future recommendations. Additionally, we expand our reflection on the significant challenges for future research in recommending renames. Hence, we answer the following research questions:

RQ1: What is the distribution of experience among developers that apply renames? We want to know how much experience developers who apply renames typically have. We use this question to understand the population from which our data has been obtained; contextualizing our data with respect to the level of experience of the developers it was generated by. This is important for future comparison with our dataset.

RQ2: What are the refactorings that occur more frequently with identifier renames? With this question, we aim to understand which types of refactorings tend to occur before or after a rename. Our assumption is that the changes made to code immediately before or after a rename have a relationship with the rename itself.

RQ3: To what extent can we use refactoring occurrence and commit message analysis to understand why different semantic changes were applied during a rename operation? Using our refactoring co-occurrence data from RQ2, we add in commit message data in an effort to see how effectively we can pinpoint the development reason for certain changes (e.g., using more general words) to words in identifier names.

RQ4: What structural changes occur when an identifier and its corresponding type are changed together? When an identifier is renamed in tandem with its data type, it may indicate a behavioral or semantic change since modifying the type ultimately may mean that the amount, or type, of data represented by an identifier has changed. This question explores structural changes made to an identifier to understand how type names are included in/removed from identifier names and changes to types affect structural changes made to identifier names.

RQ5: What semantic changes occur when an identifier and its corresponding type are changed together? When an identifier is renamed in tandem with its data type, it may indicate a behavioral or semantic change since modifying the type ultimately may mean that the amount, or type, of data represented by an identifier has changed. We explore how identifier and type naming semantics evolve together in this question, including how the plurality of identifier names correlate with collection types and whether there is a covariant or contravariant relationship between semantic updates to type names and identifier names.

RQ6: What refactorings most frequently appear before and after an identifier and its corresponding type are changed together? Are there specific semantic changes which correlate with these refactorings? This question helps us understand if there is a common set of refactoring operations applied before or after renames that involve an identifier name and its corresponding data type. Further, we explore if these refactorings imply different semantic changes to the identifier name.

5.2 Methodology

Our methodology consists of two stages - Data Collection and Detection. The Data Collection stage consists of constructing our dataset while the Detection stage consists of examining and querying the dataset for specific characteristics to help answer our research questions. Figure 5.1 represents an overview of the approach used to conduct our experiments. In the subsequent subsections, we explain in detail the approach for each activity. Due to performance requirements associated with this volume of data mining and data analysis, the activities associated with both phases were performed on a dedicated virtual machine with 16 GB of RAM, and a 3.40 GHz i7 CPU. With this configuration, the Data Collection Stage took approximately four weeks to complete, while the Detection Stage was completed in around 1.5 weeks.

5.2.1 Data Collection Stage

Projects: A key element to an empirical research study is the relevance of the dataset on which the study is based. To obtain a viable dataset, we select 800 random, open-source Java projects hosted on GitHub. These projects are part of a curated dataset of engineered software projects made available by [172]. The authors of this dataset classified engineered software projects based on the project's use of software engineering practices such as documentation, testing, and project management. For each of these 800 projects, after cloning the project repository, we enumerate over the commit log of each project to extract metadata associated with each commit. The extracted data includes the author (name and email) who was responsible for the original creation of the commit, the creation timestamp of the commit, and the names of the files that were part of the commit.

Refactorings: To obtain the set of refactorings from each project, we utilize RefactoringMiner [218]. At the time of our study, RefactoringMiner can detect 28 different refactoring operations.

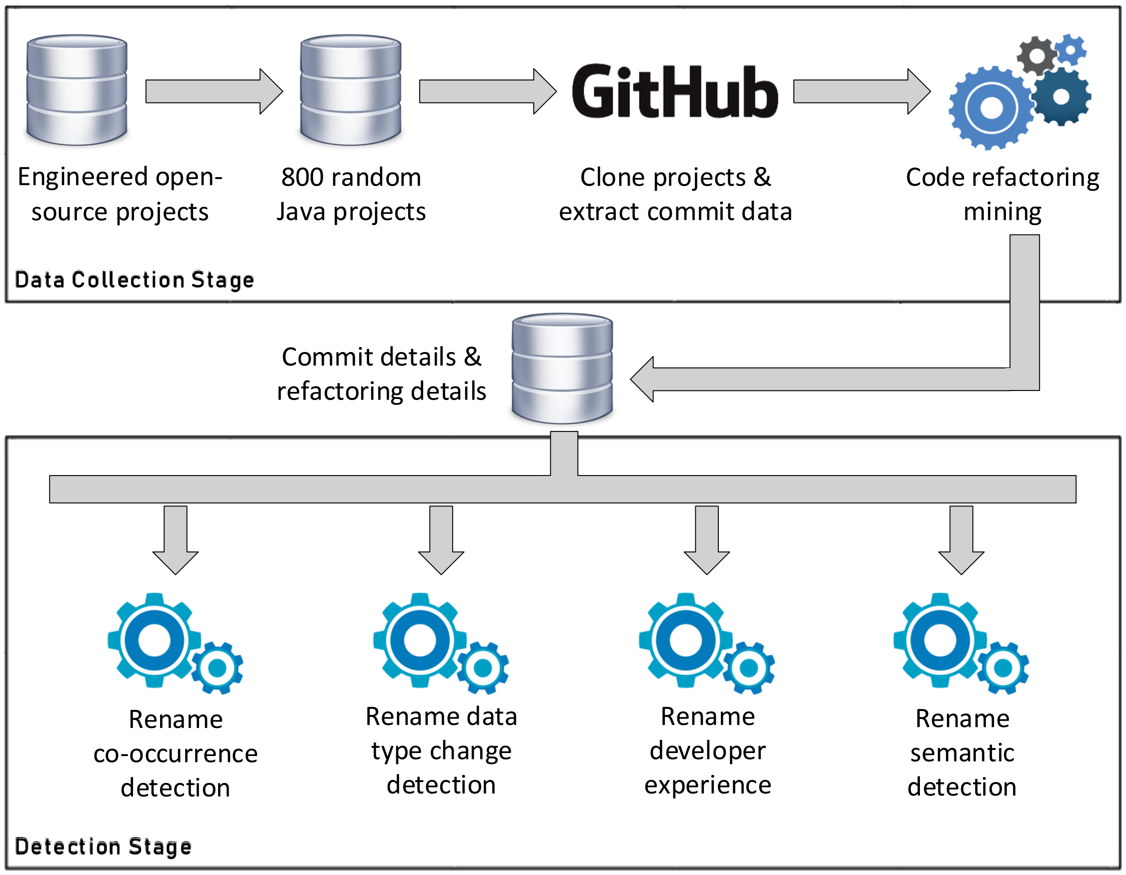

Engineered open-source projects
800 random Java projects
GitHub
Clone projects & extract commit data
Code refactoring mining
Data Collection Stage
Commit details & refactoring details
Rename co-occurrence detection
Rename data type change detection
Rename developer experience
Rename semantic detection
Detection Stage

Figure 5.1: Methodology overview

From this list of operations, seven are rename based operations. At a high-level, we utilize RefactoringMiner to iterate over all commits of a repository in chronological order. During each iteration, RefactoringMiner compares the changes made to Java source code files in order to detect refactorings in the code based on a pre-defined set of refactoring rules. While there are a few other tools that can mine refactoring operations [214], we selected RefactoringMiner since it represents state of the art in the field of refactoring detection [219], along with a precision of 98% and a recall of 87% [208,218]. Therefore, it is well suited for our large-scale mining study. We investigate the renaming operations on five types of identifiers - classes, attributes (i.e., class-level variables), methods (including getter and setters), method parameters, and method variables. Furthermore, we conduct our experiments on the entire commit history of the project (and not on a release-by-release comparison).

5.2.2 Detection Stage

Rename Forms & Semantics: We utilize the tool from one of our prior studies [191] for the detection of rename-based form and semantic updates made to an identifier's name. The tool follows the rules specified by Arnaoudova et al. [101] to determine the type of form and semantic change an identifier name undergoes when renamed. Input for the tool is the pair of old and new names associated with a renamed identifier.

First, from the output provided by RefactoringMiner, we extract all rename-based refactoring operations. Next, from these operations, we extract: 1) each pair of old and new names, 2) the name of the source code file containing the renamed identifier, 3) the name of the class containing the renamed identifier, and 4) the unique ID of the commit associated with the refactoring.

Since most identifier names are composed of multiple terms, a pre-requisite to performing the form and semantic analysis is the splitting of each name into its constituent terms. Hence, the tool utilizes the Ronin splitter algorithm implemented in the Spiral package [146] to determine the terms that form a name. The tool primarily relies on Python's Natural Language Toolkit (NLTK) [108] to compare the old and new identifier name to determine the type of semantic change made by the developer. To determine the relationship between terms in the names, the tool makes use of WordNet [168], to obtain the semantic and part of speech details about each term.

Renames With Data Type Changes: We built a custom tool to identify data types associated with identifiers that undergo a rename. Based on Java technical documentation [41], our experiments consider the following eight data types as primitives: `byte`, `short`, `int`, `long`, `float`, `double`, `boolean`, and `char`. Additionally, we examine the distribution of data types that store a group of values/references (i.e., arrays and collections) [16]. For methods, we consider the return type of the method as the data type. As `void` is not considered a type in Java [14], we exclude instances where the return type changed to/from `void`. This exclusion allowed our analysis on methods to be consistent with the other identifiers that have types– attributes, method variables, and method properties; these identifiers must be associated with a data type (either primitive or non-primitive) and thus cannot have void as their type. However, for all other instances, we apply the same processing we performed on the other identifiers in our experiment.

Our study of data type changes and their involvement in rename refactorings is limited to attributes, methods, method parameters, and method variables since classes do not have types. For each rename instance of these types of identifiers, we extract the name of the data type associated with the old and new name of the identifier. For example, the *Rename Attribute* refactoring `long connTimeToLive` → `TimeValue timeToLive` also contains a change in data type. In this instance, the developer changes the type of the identifier from `long` to `TimeValue` when renaming the attribute from `connTimeToLive` to `timeToLive`.

Rename Co-occurrence With Refactorings: We built a custom tool to identify refactorings that occur before and after rename refactoring. The tool functions by iterating over the commits which contain refactorings in our dataset. This is done in chronological order (based on the author-date – the date the commit was originally made). Since our rename refactorings are related to classes, attributes, methods, method parameters, and method variables, we restrict our detection to refactorings that are applied to only these types of identifiers. For each renamed identifier type, we first extract all unique instances. Next, we iterated through all refactorings searching for refactorings that involved the specific instance. Our process does not take into account the time duration between commits when looking for surrounding refactoring commits.

To better highlight this process, consider the example where we detect the class `stormpot.CountingAllocatorWrapper` as being renamed to `stormpot.CountingAllocator` [75]. We first query our list of unique attributes, methods, parameters, and variables for identifiers that were part of this class and had also undergone a refactoring. Our search results in an attribute, `counter`, belonging to this class, which had undergone a rename refactoring (prior to the class being renamed) [76]. We utilize the author-date attribute associated with a commit to determine the order of the commits. Finally, we record this pair of refactorings in our database. It should be noted that the version of RefacotringMiner we utilize only supports rename refactoring operations for parameters. Hence, we did not obtain other types of refactoring operations that developers might apply to parameters.

Rename Co-occurrence With Data Type: The purpose of this activity is to detect and analyze the co-occurrence of rename refactorings that also contain a data type change to the renamed identifier. Hence, we follow an approach similar to the general rename refactoring co-occurrences described above. However, in this new approach, we limit the dataset to only rename instances with a data type change; the general approach did not consider data type changes. For example, the *Rename Attribute* refactoring `HistoryMap historyMap` → `History history` contains a data type change from `HistoryMap` to `History` [10]. However, before performing this rename, the developer performs a *Pull Up Attribute* refactoring operation on the attribute [11]. In this instance, the refactoring operations *Rename Attribute* and *Pull Up Attribute* co-occur when the data type of the attribute changes during its rename.

Commit Log Analysis: To derive the developer's rationale for performing a rename, we look at the commit log as a means of contextualizing the rename. Hence, our experiment involves the performance of a topic modeling and n-gram analysis of commit messages. For our topic modeling analysis, we utilize the Latent Dirichlet Allocation (LDA) [109] algorithm. Additionally, we use a combination of topic coherence [202] and manual empirical analysis as a means to determine the ideal number of topics; past research has shown that the number of topics can vary between studies and datasets [104]. A prerequisite to these activities was a text preprocessing task where we cleansed and normalized the commit messages. Normalization is a process of transforming non-standard words into a standard and convenient format [151]. Some key steps in our preprocessing include: removal of stopwords, URLs, numeric and alphanumeric characters/words, and non-dictionary words. Additionally, we also expand contractions (e.g., 'I'm' → 'I am') and perform stemming and lemmatizing on words.

Taxonomy: Additionally, we perform a qualitative analysis on the source code changes that accompany rename refactorings. In this experiment, we manually review the diff of the commit in order to understand if the rename was made in conjunction with other changes to the code or by itself. As a setup for this experiment, we select 30 random rename instances from each of the five types of identifiers. This results in a total of 150 source code files for our manual review. When performing the review, the reviewers first examine changes made by the developer to surrounding code elements and the commit message. Next, the reviewers determine the rationale for the rename. Finally, the reviewers compare their individual taxonomy annotations and agree on a final set. The reviewed source files were then annotated using this finalized taxonomy.

Developer Experience: The purpose of this activity is to determine the experience of the developers that refactor the source code in a project. As our study is on renames, we derive the experience of developers where the developers refactoring operations are limited to only renames, developers who perform all refactoring operations, and developers who perform only non-rename refactoring operations. As this is a large empirical study, obtaining the experience of each developer, associated with a project, in our dataset is not feasible and can also be subjective. Hence, to overcome this challenge, we perform a more objective-based experiment where we follow the approach utilized by [153]. In their approach, the authors use project contribution as a proxy for developer experience within a project. Hence, for each developer in each project, we calculate the Developer's Commit Ratio (DCR). This ratio measures the number of individual commits made by the developer against all project commits. In other words, DCR = $(\frac{IndividualContributorCommits}{TotalAppCommits})$. We utilize the project's commit log along with the output of RefactoringMiner to determine the developers that belong to each of the three groups. Using details in the commit log, we first calculate the DCR for all developers in a project. Next, using the output of RefactoringMiner, we split the developers into their respective groups based on the type of refactoring operations they had performed during the lifetime of the project. To mitigate the threat of misattributing commits due to the use of GitHub features such as pull requests, we only consider the author of a commit as its developer.

5.3 Experimental Results

In this section, we discuss the results of our experiments. The discussion is broken down into six Research Questions (RQs). While RQs 1-3 focus on all rename refactorings, RQs 4-6 focus specifically on rename refactorings in which the renamed identifiers also had a change in data type. The RQs are designed to help us understand how data type changes affect the evolution of identifier names when these changes are applied in tandem. In RQ1, we focus on the experience of developers that perform rename refactorings versus other types of refactoring operations. In RQ2, we discuss what types of refactorings occur before or after a rename refactoring. Additionally, we look at how often rename refactorings are preceded or followed by another refactoring, and what types of refactorings these preceding or following changes represent. In RQ3, intending to contextualize identifier renames, we combine and discuss data from RQ2 with commit message information and the semantic change types. Our end goal is to utilize the commit message and refactoring information to contextualize the semantic change types we detected in our set of renames. In RQ4, we examine the structural changes applied to an identifier name when both it and its corresponding type are changed together. In RQ5, we apply similar analysis as in RQ4 except we look at semantic, instead of structural, changes; identifying how the meaning of identifier names evolve when their type is changed in-tandem. Finally, RQ6 is similar to RQ2, but we focus on refactorings surrounding identifier renames which include a change to the corresponding type.

Table 5.1: Distribution of the top five refactorings

Refactoring Type	**Count**	**Percentage**
Rename Attribute	137,842	19.37%
Rename Variable	84,010	11.81%
Rename Method	82,206	11.55%
Move Class	76,265	10.72%
Extract Method	47,477	6.67%
Others	*283,695*	*39.87%*

5.3.1 Data Summary

For context, we present a summary of our dataset before we discuss our results.

First, with regards to project cloning, in total, we collected 748,001 commits with a project containing 732 commits and 19 developers on average. In terms of recentness, the projects were cloned in early 2019, and approximately 74.6% of the projects had their most recent commit within the last four years. Next, looking at the RefactoringMiner output, we identified 711,495 refactoring operations, with each project in our dataset exhibiting more than one refactoring operation. After the removal of outliers (via the Tukey's fences approach), on average, each project had 450.8 refactoring operations performed by seven developers. Approximately 53.51% of the refactoring operations in our dataset were rename based. We present the top five refactoring operations, from our mined dataset, in Table 5.1.

Looking at the form type and semantic updates data, obtained during the Detection stage (Section 5.2), we observed that developers more frequently perform a Simple form type rename compared to Complex, Formatting, and Reordering. In terms of semantic updates, most identifiers undergo a change in meaning, with a narrowing in meaning occurring the most. Shown in Table 5.2 is the distribution of rename form and semantic meaning types that were performed by all developers in our dataset.

From our analysis of renames with data type changes, approximately 17.39% (53,962) of renames were performed with a change in data type. From this set, developers frequently change the type of method variables followed by method parameters. A breakdown into the individual identifier types is presented in Table 5.3. Out of the 800 projects in our dataset, 769 ($\approx$ 96.13%) of these projects exhibited rename instances that had a change in data type. Looking at the individual identifier

Table 5.2: Distribution of rename forms and semantic meaning updates made to identifier names by developers

Type	**Count**	**Percentage**
Rename form types		
Simple	259,754	68.31%
Complex	109,860	28.89%
Formatting	8,916	2.34%
Reordering	1,732	0.46%
Rename semantic meaning updates		
Preserve	29,568	7.78%
Change	350,694	92.22%
Change – Narrow		44.21%
Change – Add		37.93%
Change – Broaden		15.09%
Change – Remove		2.58%
Change – Antonym		0.19%

Table 5.3: Distribution of rename-based type changes

Type Changed	Type of Rename	Count	Percentage
No	Rename Attribute	128,486	41.41%
No	Rename Variable	61,665	19.87%
No	Rename Method	37,923	12.22%
No	Rename Parameter	28,086	9.05%
Yes	Rename Variable	21,885	7.05%
Yes	Rename Parameter	16,285	5.25%
Yes	Rename Attribute	9,355	3.01%
Yes	Rename Method	6,397	2.06%
No	Move And Rename Attribute	187	0.06%
Yes	Move And Rename Attribute	40	0.01%

types, approximately 80.25% of all projects in the dataset had an attribute rename with a change in data type, while approximately 73.65%, 92.25%, and 81% of projects had a rename of a method, variable, and parameter occurring in tandem with a data type change respectively. Furthermore, approximately 42.75% of the projects from our dataset of 800 contained a refactoring occurring either before and/or after a rename refactoring that also contained a data type change.

Finally, we followed [101]'s approach to identify documented renamings in our dataset. From the set of mined rename refactoring commits, approximately 6.9% (or 4,701 out of 68,121) of the commits documented the renaming, compared to less than 1% in [101]'s dataset. This means that most commit messages do not explicitly discuss the rename operation. However, while renames are not always documented, the motivation behind the rename may still be gleaned from the commit message (e.g., the commit may discuss clean-up, bug fixing, changing a method's behavior). Not all commit messages which document renames specify why the rename is needed (e.g., "renaming some variables" [21]) and, likewise, some rename motivations can be found in commit messages which do not mention the rename itself (e.g., "extract method to convert db entity to generic entity" [36]). This percentage does indicate a potential need for rename documentation support.

5.3.2 RQ1: What is the distribution of experience among developers that apply renames?

To compare the distributions of DCR for developers who had performed only renames, only non-renames, and a mix of rename and non-rename refactorings, we follow the same approach as [153].

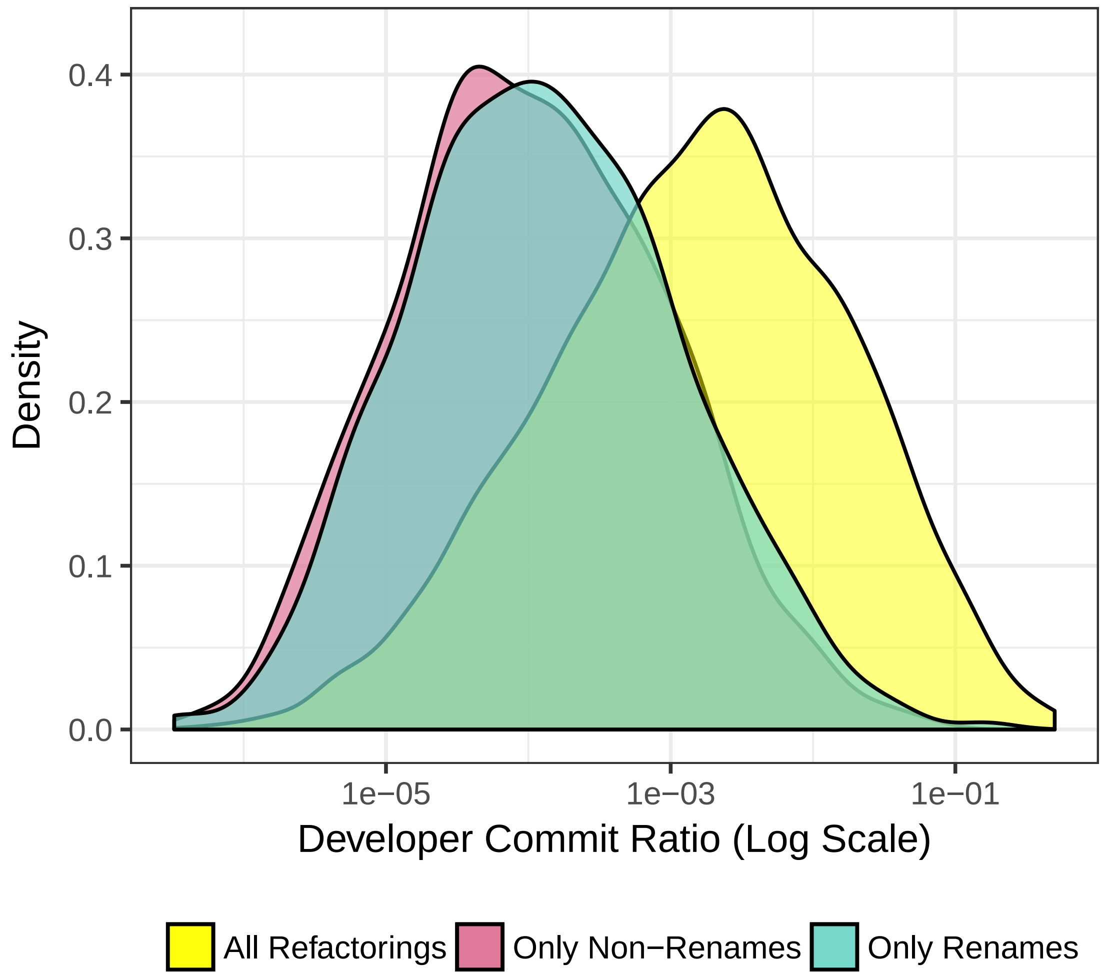


Figure 5.2: Distribution of DCR values for developers based on the type of refactoring performed in their project

Since the number of developers in each project differs, we calculate an adjusted DCR value for each developer by dividing the developer's original DCR value by the number of developers in the project. We also restrict our experiment to projects that had only two or more developers. Figure 5.2 depicts the distribution of DCR values for developers based on the type of refactoring performed in their project.

Our observation of developers who perform all types of refactorings having a higher DCR than those that perform only rename refactorings is in line with the research indicating that rename operations are considered simpler, or more accessible, compared to other refactoring operations.

That is, developers who are less experienced feel more comfortable applying them [156, 170, 224]. However, it is interesting that developers who perform only renames share a similar DCR value as those that perform only non-rename refactorings. To further validate these findings, we perform a nonparametric Mann-Whitney-Wilcoxon test on the DCR values for developers that belonged to these categories. We obtained a statistically significant p-value (< 0.05) when the DCR values of developers who performed only rename refactorings were compared to developers that perform all types of refactorings. This value confirms that developers that contribute less to a project are more likely to perform rename refactorings, which are generally considered easier to apply due to wide IDE support despite developers also generally agreeing that renaming is a difficult problem [101].

Looking at the different types of identifier rename forms, we observe that there is no significant difference in the distribution of renaming forms between developers that perform only renames and those that perform all types of refactorings. Similarly, the types of semantic updates to an identifier name also showed no significant differences among these two groups of developers. Table 5.4 provides a breakdown of the distribution of rename form and semantic meaning updates based on developer type. Our experiment on developer experience shows that developers with more project experience (i.e., contributions) are more accustomed to performing a multitude of different types of refactoring operations. This is not surprising as these developers have more experience and knowledge of the codebase (and system) and would be more comfortable in implementing design/structural changes to the project. Given that rename refactorings have broad IDE support and are syntactically simple modifications, inexperienced developers will naturally be drawn into making such refactorings in the project.

Summary for RQ1: Developers with limited project experience are more inclined to perform only rename refactorings than other types of refactorings (which may alter the design of the system). This is an important context for any future recommendation effort and particularly for our data. Given that many of the developers performing the renames we analyzed have less experience on average, our results may reflect this lack of experience. Further research is needed to confirm the connection between the quality, of renames and developer experience.

5.3.3 RQ2: What are the refactorings that occur more frequently with identifier renames?

To derive the extent to which non-rename refactorings can either influence or be influenced by a rename, we study *the type of refactoring commits that occur just before and after a rename refactoring commit.* This is based on the idea that renames are likely to occur with other refactorings;

Table 5.4: Distribution of rename form and semantic meaning updates split by developers who performed all refactoring operations and those that performed only rename refactoring operations.

Type	**Only Renames**	**All Refactorings**
	Percentage	**Percentage**
	Rename form types	
Simple	64.65%	67.01%
Complex	30.55%	29.96%
Formatting	4.56%	2.52%
Reordering	0.24%	0.51%
	Rename semantic meaning updates	
Preserve	9.97%	8.50%
Change	90.03%	91.50%
Change – Narrow	48.99%	48.08%
Change – Add	29.93%	32.68%
Change – Broaden	18.33%	16.46%
Change – Remove	2.58%	2.56%
Change – Antonym	0.17%	0.21%

Table 5.5: Top 3 refactoring operations that occur before a class, attribute, method and method variable are renamed

Refactoring Operation	**Count**	**Percentage**	**Commit Message Key Terms**
Refactoring operations before a class rename			
Move Class	3,069	26.96%	package, structure, change
Rename Method	2,062	18.12%	code, clean, change, fix
Rename Variable	1,376	12.09%	add, code, test, support
Others	*4,875*	*42.83%*	*N/A*
Refactoring operations before an attribute rename			
Move Attribute	1,499	83.32%	added, fix, support, test
Pull Up Attribute	220	12.23%	added, simplification, extract
Push Down Attribute	73	4.06%	separate, remove, added
Others	*7*	*0.39%*	*N/A*
Refactoring operations before a method rename			
Rename Method	1,760	19.58%	revert, implementation, test
Extract Method	1,666	18.53%	fix, added, modified, test
Rename Variable	1,364	15.17%	added, test, fix, change
Others	*4,201*	*46.72%*	*N/A*
Refactoring operations before a method variable rename			
Rename Variable	3,067	90.66%	revert, added, test, fix
Extract Variable	305	9.02%	added, string, test, fix
Inline Variable	6	0.18%	fix, working, change
Others	*5*	*0.15%*	*N/A*

an assumption supported by Murphy-Hill et al. [173] who shows that developers perform renames in batches more so than other refactorings and, most often, that refactorings occur on multiple related code elements. This part of our study focuses on the renames of classes, attributes, methods, method parameters, and method variables. For each entity type, we extract the list of unique instances that underwent a rename and then search for the refactoring that directly precedes and directly follows (i.e., there may be non-refactoring commits that we skip) the rename for either the same entity or child entities (as in the case of classes and methods).

Interestingly, we observe that for all elements that are subject to renames, developers frequently perform the rename in isolation with respect to other refactorings. In other words, approximately 91.97% (or 349,731) of rename commits had no refactorings occur one commit before or one commit after. However, this does not mean that rename is the only action applied to this element during its lifetime. Upon the inspection of some cases, there were changes, applied to the element, which are not considered refactoring (e.g., adding lines of code to a method, adding a given identifier as a parameter to another method). For scenarios where there are refactorings either before or after a rename, we noticed that more operations occur before a rename ($\approx 6.27\%$) than after ($\approx 1.73\%$).

In general, the majority of the refactorings that occur before a rename are related to changes/updates to functionality. Additionally, we observe that some of these commits are bug fix related or due to developers either adding or updating unit test files. For example, in order to include new functionality, a developer refactors the existing code by creating a new method called `getClassURL` by performing an *Extract Method* operation [50]. Thereafter the developer renames the newly created method to `getClassUrl` to ensure that name follows “Google’s style rules” [49].

Even though the number of refactorings occurring after a rename is much smaller, we did notice that most of these refactorings are associated with some form of code reversal/reverting. As an example, a developer initially renames a method from `getIncludedPublishers` to `getEnabledSources` when introducing new functionality [51]. However, in a subsequent commit [52], the developer removes this functionality from the method and also reverts back to the original method name.

As the majority of refactoring operations occur before a rename, in the following subsections, we drill-down into each element type with the aim of discovering the common types of refactorings that precede the renaming of the element and also the extent to which the commit log can contextualize the relationship between these refactorings. Table 5.5 highlights the distribution of the top three refactoring operations that occur before a class, attribute, method, and method variable is renamed. Also provided in this table are the common terms we extracted from our topic-modeling and n-gram analysis of the commit messages that are associated with these refactoring operations.

Class Rename

Our study of class renames involve identifying the refactorings performed on the class and all elements within the class (i.e., attribute, methods, method parameters, and method variables) immediately before and after the developer renames the class. We observe that developers more frequently performed a *Move Class* refactoring before renaming the class. Results from our topic modeling and n-gram analysis coupled with a manual analysis of random messages showed that activities related to restructuring project structures and change of package names cause developers to rename class names. For example, in [24] a developer moves the class `BasicAuthLoginCommand` from `com.heroku.api.command` to `com.heroku.api.command.login` with the message "reorganized commands into appropriate packages." The next refactoring operation [25] performed on this class is renaming the class to `BasicAuthLogin`. The reason for the rename is "...to simplify some of the names."

Looking at the number of non-refactoring commits that separate a *Move Class* from a *Rename Class*, we observe that the majority of renames ($\approx 7.15\%$) occur in the commit immediately following the move. We also observe that a majority of these pairs of refactoring commits fell within one to five commits of each other – approximately 27.73% of the time. This lends support to the idea that they are related; *Move Class* refactorings are frequently done near the same time as *Rename Class.* While further investigation is required to determine when it is appropriate to recommend a rename in this situation, our data highlights this relationship as a good avenue for future, deeper research into what indicates that the rename will be performed versus when it will not be.

Attribute Rename

Similar to classes, developers perform move operations on attributes before renaming them. Looking at the commit messages, change in functionality (specifically adding of new features) is one of the most common reasons developers move an attribute. As an example, in commit [19], the developer moves the attribute `String jobId` with the message "added the jobId to a few more logs". The subsequent refactoring commit [18] for this attribute involves a renaming operation in which the attribute is renamed to `context` as part of a "cleanup" activity. We observe that around 71% of the renames occur in the commit immediately after the developer moves the attribute. Additionally, around 82% of rename refactorings take place within five commits after the *Move Attribute* operation.

Method Rename

For methods, we investigate the refactorings that are applied to the method and its members (i.e., parameters and variables) just prior to and after the method is renamed. Interestingly, we observe that developers perform a rename to the method before renaming it again more than any other type of refactoring. Based on the terms in the commit log, we observe that the reason for the initial rename is due to developers changing the behavior/purpose of the method. Furthermore, we notice that the second occurrence of the method rename reverts the first rename operation. For example, in [43], the developer renames the method `showDelivery` to `showOwnDelivery` as part of a functionality change, with the commit message "Minor changes to access controls in instructor MVC". In the subsequent commit [44], the developer reverts the name change as part of cleanup activities with the message "Final tidy of older instructor MVC".

Looking at the interval between commits, the majority ($\approx$ 15.22%) of the method-rename pairs of refactorings occur one after another. Further, a gap of between 1 to 5 commits occurs around 37.68% of the time between two method renames.

Method Variable Rename

Like methods, method variables also undergo rename operations in succession. Once again, looking at the commit messages, we observe that the reason for the initial rename tends to be due to either refactoring or change (including reversals) in functionality. It is also interesting to note that the developers revert the variable name of the initial commit in the next rename. For example, in [54] the developer renames the variable `drop` to `assembledDrop` with the message "simplified drop assembly a bit". The next commit [55] reverts the variable name when the developer performs a "misc code cleanup" activity. Finally, a gap of between 1 to 5 commits occurs around 33.94% of the time between two variable renames.

Summary for RQ2: We show that in most scenarios, renaming of an element does not generally seem to be influenced by, nor does itself influence another type of refactoring on the same element. This indicates that an analysis of non-refactoring operations will be required to understand how changes to code around a rename affect or are affected by the rename. However, there is a subset of renames that occur directly before or after another refactoring. Most commonly, the refactorings occurring before a rename are *Extract Method*, *Move Attribute*, *Move Class*, and *Rename*. In particular, we observe that 71% of the time, a rename occurs in the commit directly following a *Move Attribute*, and 82% of the time, this rename is within five commits after the *Move Attribute*

Table 5.6: An overview of the types of semantic updates an identifier name undergoes

Identifier Type	Refactoring Before Rename	Top 3 Types of Rename Forms	Type of Semantic Update	Top 3 Semantic Change Subtypes
Class	Move Class (Total Count: 3,160)	Simple (57.82%) Complex (34.68%) Formatting (5.54%)	Change (84.14%) Preserve (15.85%)	Narrow (63.56%) Broaden (28.13%) Add (3.65%)
	Rename Method (Total Count: 2,179)	Simple (65.26%) Complex (30.29%) Formatting (2.98%)	Change (90.0%) Preserve (10.0%)	Narrow (57.42%) Broaden (31.56%) Add (6.78%)
	Rename Variable (Total Count: 1,479)	Simple (61.19%) Complex (34.69%) Formatting (2.64%)	Change (100.0%)	Narrow (100%)
Attribute	Move Attribute (Total Count: 1,499)	Simple (67.44%) Complex (29.75%) Formatting (2.54%)	Change (94.66%) Preserve (5.34%)	Add (54.05%) Narrow (24.59%) Broaden (16.07%)
	Pull Up Attribute (Total Count: 220)	Simple (55.91%) Complex (35%) Formatting (8.18%)	Change (85.0%) Preserve (15.0%)	Narrow (66.84%) Broaden (25.67%) Add (3.21%)
	Push Down Attribute (Total Count: 74)	Simple (62.16%) Complex (28.57%) Formatting (6.76%)	Change (63.51%) Preserve (36.49%)	Narrow (70.21%) Broaden (23.4%) Add (2.13%)
Method	Rename Method (Total Count: 2,158)	Simple (66.22%) Complex (23.17%) Formatting (9.87%)	Change (81.19%) Preserve (18.81%)	Narrow (36.42%) Broaden (31.16%) Add (24.14%)
	Extract Method (Total Count: 1,694)	Simple (52.42%) Complex (43.15%) Formatting (3.96%)	Change (85.42%) Preserve (14.58%)	Narrow (64.06%) Broaden (26.12%) Add (4.49%)
	Rename Variable (Total Count: 1,387)	Simple (51.62%) Complex (43.98%) Formatting (3.89%)	Change (87.41%) Preserve (12.59%)	Narrow (49.28%) Broaden (32.86%) Remove (9.17%)
Variable	Rename Variable (Total Count: 3,067)	Simple (87.41%) Complex (12.36%) Formatting (0.23%)	Change (98.89%) Preserve (1.11%)	Add(77.35%) Narrow (14.93%) Broaden (6.17%)
	Extract Variable (Total Count: 305)	Simple (61.64%) Complex (36.72%) Formatting (1.31%)	Change (92.13%) Preserve (7.87%)	Narrow (71.17%) Broaden (19.57%) Add(6.05%)
	Inline Variable (Total Count: 6)	Simple (83.33%) Complex (16.67%)	Change (100%)	Add (66.67%) Narrow(33.33%)

operation. In other cases (*Move Class*, *Rename Method*), this percentage rests between 15 and 27%. Finally, in situations where a rename follows another rename, we observe that developers revert to the original name when performing the second rename.

5.3.4 RQ3: To what extent can we use refactoring occurrence and commit message analysis to understand why different semantic changes were applied during a rename operation?

To answer this question, we look at the types of semantic changes applied to identifier names given that another refactoring was applied in the previous commit. We then analyze this data to understand whether the refactoring that happened before the rename had any effect on the semantic change applied during the rename. Additionally, we perform an analysis of commit messages using LDA and bi/trigrams in an effort to further contextualize the semantic change; using information about why a given refactoring was applied before the rename to help us understand the semantic changes observed during renames applied afterward.

The first observation we make is that renames applied after another refactoring most frequently changed the target name's meaning somehow; the meaning was less frequently preserved. Therefore, we will first look at renames that changed the meaning of the identifier they were applied to. Table 5.6 highlights the distribution of these change types for elements that undergo a rename after another type of refactoring operation.

We observe that the majority of the name changes were related to a narrowing in the meaning of the name. Generally, a narrowing in the meaning of an identifier name is related to a specialization of functionality. For example, in commit [35], a developer created the method `readImage(width int, height int)` by performing an *Extract Method* operation in order to add "missing functionality". In a subsequent refactoring operation on this method, the developer renames the method to `readZlibImage(width int, height int)` with the message "Added read support for GM8 gmk files" [40]. As can be seen by the message, the developer specializes the method and hence reflects this behavior in the new method name by narrowing its meaning.

The next most common type of semantic change was the broadening of the identifier's name. Developers perform a broadening of the name when they generalize the behavior of the identifier. As an example, in commit [29], a developer performs a *Pull Up Attribute* on `idColumn` as part of generalizing change – "Create generic table class" . Thereafter, the developer renames the attribute to `id` in order to make it consistent with the earlier generalizing task – "Rename generic table column fields" [28]. Finally, adding to the identifier name was the third most frequent type of semantic change.

There are a few interesting things to point out in Table 5.6. The first is that a *Rename Variable* followed by another *Rename Variable* tended to add meaning instead of narrow or broaden. The same applies to renames occurring after a *Move Attribute* refactoring and after an *Inline Variable* refactoring. However, these are the only examples of a break from the typical pattern of Narrow being the most common semantic change type. If we only contextualize using refactorings applied before renames, there are few significant differences in the types of semantic changes applied after different types of refactorings. While this data does indicate the popularity of narrowing, adding to, or broadening the meaning of a name, it does not completely help us understand what the developers were trying to accomplish; an *Extract Method* refactoring occurring before a rename does not serve as a strong indicator of what semantic change will happen if a rename is applied afterward.

To help us further contextualize these refactorings and the renames occurring afterward, we perform LDA and n-gram analysis on commit messages associated with the rename refactorings occurring after a refactoring operation. Our previous work also used LDA in a similar context [191], but it performed LDA analysis on the commit message associated with the rename without taking into account if the rename occurred in isolation or immediately after another refactoring. We extend the topic modeling approach in [191] by incorporating additional text preprocessing and the use of topic coherence scores in order to improve the quality of our text analysis compared to the original study. The results of this analysis are in Tables 5.7, 5.8, 5.9, and 5.10. In each table, we show the two strongest topics from LDA along with either a bigram or trigram analysis. We present either the bigram or trigram that is the most relevant. Using the data in these tables, we can see some indication of what development activity caused different types of semantic changes when applying a rename.

Table 5.7 shows data for all method renames that are preceded by a variable rename, and resulted in the name of the method broadening in meaning. These preceded a rename which resulted in a *broaden meaning*. The data here indicates changes to a model and changes to a factory. An analysis of the commit messages associated with these topics shows that the updates are due to bug fixes or code optimizations. For example, in commit [70], the broadening of the name is associated with the message "...Made the factory generic", which a broaden meaning rename would logically follow. Table 5.8 has similar data but for a set of *Extract Variable* refactorings which preceded a *narrowing* of the identifier name meaning via rename. The topics and bigrams here indicate code related to data binding, code updates, and code fixes. Again, we took a look at the commit messages associated with this data and found that most of the data bindings were specific to a certain project in our corpus. In this instance [37], the developer uses a generic message, "Updated data binding code...". Ignoring this set of commits, a majority of the remaining messages were associated with bug fixes.

Table 5.7: Broadening of a method name after a variable rename

Analysis	**Output**
LDA Topic 1	change (0.090), model (0.086), past (0.068), discussed (0.068), allow (0.019), lambda(0.019), route(0.019), work(0.015), early(0.014), simplified(0.014)
LDA Topic 2	change (0.088), model (0.061), past (0.049), discussed (0.049), fix (0.028), factory (0.026), changed (0.023), loader (0.023), add (0.017), set (0.013)
Trigram	(discussed, past, model), (change, discussed, past), (model, change, discussed), (past, model, change), (discussed, past, added), (changed, loader, factory), (loader, factory, changed), (factory, changed, loader), (location, model, change), (render, nicely, html)

Table 5.8: Narrowing of a variable name after its extraction

Analysis	**Output**
LDA Topic 1	code (0.091), binding (0.083), data (0.081), updated (0.074), fix (0.028), add (0.025), support (0.017), cr (0.009), custom (0.009), request (0.009)
LDA Topic 2	code (0.067), updated (0.060), binding (0.059), data (0.058), record (0.013), id (0.010), custom (0.010), introduced (0.010), remove (0.010), cr (0.007)
Bigram	(data, binding), (binding, code), (updated, data), (code, updated), (revamped, hibernate),(added, method), (array, fix), (attribute, handle), (binding, warning)

Table 5.9: Narrowing of an attribute name after its pulled-up

Analysis	Output
LDA Topic 1	work (0.094), introduce (0.043), security (0.034), option (0.034), addition (0.034), start (0.018), add (0.018), took (0.018), thread (0.018), ongoing (0.018)
LDA Topic 2	symbol (0.077), table (0.077), work (0.061), unit (0.031), option (0.031), property (0.024), fixed (0.022), hierarchy (0.016), added (0.016), implementation (0.016)
Trigram	(hierarchy, option, reduce), (implemented, hierarchy, option), (option, reduce, code), (reduce, code, duplication), (code, duplication, implemented), (duplication, implemented, hierarchy), (gross, value, gross), (addition, security, addition), (code, added, support), (entity, id, field)

Table 5.10: Adding meaning to a class name after moving it

Analysis	Output
LDA Topic 1	method (0.189), added (0.083), adding (0.072), increased (0.071), incremental (0.071), stub (0.071), anonymous (0.071), truly (0.071), fix (0.013), subset (0.013)
LDA Topic 2	test (0.198), validation (0.043), removing (0.030), enable (0.029), mapping (0.029), upgrade (0.029), failing (0.029), concept (0.029), collection (0.015), contains (0.015)
Trigram	(added, method, adding), (adding, truly, anonymous), (incremental, stub, method), (method, added, method), (method, adding, truly), (stub, method, added), (truly, anonymous, increased), (anonymous, increased, incremental), (cleaned, scorer, removing), (field, tree, context)

Table 5.9 shows the implementation of options and reduction in code duplication which preceded a *narrowing* in meaning. An analysis of the commit messages associated with this table shows that the removal of duplicate [83] and legacy [38] code is a task associated with code cleanup activities. These activities can also range from simple identifier renames [42] to more intensive structural changes [39]. Finally, Table 5.10 indicates the addition of new methods associated with moving a class to a different location, which preceded an *add meaning* change. Examining these commit messages reveals that methods are added in response to enhancing the existing design of the system after the class is moved and hence contribute to the renaming of the class, such as in the case of [1], where the developer performs a "...Method grouping" in the newly moved class.

Preserve meaning was the least occurring semantic type, and not surprisingly, the frequently occurring terms in these commit messages were not change related. These terms include 'fix', 'test' and 'work'. Generally, such terms are associated with behavior correction. Hence, developers feel that the update they make to the code does not necessarily deviate from the originally expected behavior of the identifier. For example, in [7] as part of updates to the user interface, the developer performs a *Pull Up Method* operation on the method `calcTotal`. The next update [6] to this method is to address an issue, and as part of this task, the developer renames the method to `calculateTotal` to better represent its intended behavior. A cursory glance at the method shows no changes to the functional behavior exhibited by this method.

Summary for RQ3: Developers frequently change the semantic meaning of an identifier name when performing a rename after a refactoring, rather than preserving it. Most frequently, a rename will change this meaning by narrowing (i.e., specializing) the identifier name it is applied to. While the rationale for some semantic changes can be derived from the commit log in addition to the actions that occurred just prior to the rename, classical ways of analyzing large numbers of commit messages provide only a high-level understanding of this rationale and require significant manual analysis to help us fully understand the rationale. The answer to this RQ is that refactorings, occurring before and after a rename, and commit messages can give us some high-level insight into how names semantically change and why. Still, our data shows that further research using additional software artifacts, and new methods of natural language text analysis for software engineering, are required to provide us with stronger insights.

Table 5.11: Distribution of identifier form types when a change in data type occurs

Form Type Change	**Count (Total: 53, 962)**	**Percentage**
Simple	32,448	60.13%
Complex	21,169	39.23%
Formatting	314	0.58%
Reordering	31	0.06%

5.3.5 RQ4: What structural changes occur when an identifier and its corresponding type are changed together?

From our analysis of 310,309 identifier rename instances, we observe that 17.39% (or 53,962) of identifier renames involve a change to their corresponding type. We are interested in understanding how changes to the type name correlate with modifications to the structure of an identifier name. A breakdown of renames which included a change in data type is shown in Table 5.3.

First, we look at rename forms (i.e., Simple, Complex, Reordering, and Formatting). A Simple rename involves a change of a single term between the old and new name of the identifier (e.g., `char[] password` → `byte[] encodedPassword`). The *Rename Variable* refactoring operation `String moveCoords` → `Point point`, on the other hand, falls under a Complex rename as more than one term between the old and new identifier name has changed. Reordering involves a change of position of terms (e.g., `RecordId recordId` → `IdRecord idRecord`), while Formatting is due to change of case or the addition/removal of a special character (e.g., `AbstractDropDown dropdown` → `DropDown dropDown`). Looking at the types of rename forms, as shown in Table 5.11, we observe that approximately 60.13% of data type changes are associated with Simple changes to the identifier's name, while Complex changes account for approximately 39.23%. Formatting and Reordering changes each account for less than 1%.

Additionally, we investigate the extent to which an identifier's name contains the name of its data type to see if the type is generally added or removed as identifiers are changed. Prior work considers the inclusion of a type name in the associated identifier's name as an impediment to software maintenance and code comprehension activities [164]. As some insight into this, it could be argued that, in strongly typed languages, including the name of a type in an identifier's name is redundant due to the type being explicitly present already, and modern IDEs will generally inform the developer of an identifier's type using annotations. Another drawback of this naming approach is that the developer will be forced to rename the identifier when changing its data type (or risk the name becoming out of date). This might be a substantial number of instances throughout the codebase; not just the statement declaring the identifier. For example, when naming the variable `String tupleString`, the developer appends the name of the data type, `String`, to the identifier's name. For this specific example, we observe that the developer, in a subsequent commit, renames the identifier to `List<String> tuple`. As such, it can be seen that the developer had to adjust the name of the identifier due to the old data type being present in the name.

For each instance of a rename refactoring in our dataset, we check if the old and new name contains its respected data type as part of its name (i.e., the identifier name either starts with, ends with, contains or is an exact case-insensitive match of the name of the data type). First, looking at all rename instances (i.e., renames with and without a data type change), we observe that approximately 83.69% (out of 310,309) of the rename refactorings did not contain the name of the data type as part of the old or new identifier's name. Within this 83.69% of renames, approximately 10% of these renames had a change in data type, while the remaining 90% retain the same data type.

Focusing on the remaining 16.31% (or 50,621) of renames on identifier names which contain the name of their corresponding data-type, we have two groups which are summarized in Table 5.12: G1) identifiers whose corresponding data type changed (top half of the table with 26,227 rename instances); and G2) identifiers whose corresponding data type did not change (bottom half of the table with 24,394 rename instances). Identifier names in G1 tended to exactly match the name of their type even after being renamed (e.g., `LocationStrategy locationStrategy` → `ElementLocator elementLocator`) 34.86% of the time and, when the name of the type was not originally present, they tended to be changed to exactly match their type during a rename (e.g., `BitRateType bitRate` → `BitRateType bitRateType`) 18.73% of the time. This indicates that when a data type and identifier name are changed in-tandem, there is a tendency to include (or keep) the name of the type within the identifier name.

Identifier names in G2 are similar in that the majority of most frequent cases involve adding (or keeping) the data type name to (in) the identifier name. The primary difference between G1 and G2 is that G1 identifiers tend to be exact matches; the identifier name and type name are exactly the same. In G2, the type names are most frequently appended to the end of the identifier name; the type name is a substring of the identifier name. An explanation for this difference may be that, since types in G1 were modified in-tandem with identifier names, the identifier names are more intricately linked to the type name. Either by already having included it (in the 34.86% case) or

Table 5.12: Distribution of occurrence for the different scenarios where the name of the data type is present in the identifier's name

Old Identifier Name	**New Identifier Name**	**Count**	**Percentage**
Renames with data type changes that contain the name of the data type in the identifier's name (Total Count: 26,227)			
Exact match	Exact match	9,143	34.86%
Does not contain	Exact match	4,913	18.73%
Exact match	Does not contain	3,746	14.28%
...other combinations		8,425	32.12%
Renames without data type changes that contain the name of the data type in the identifier's name (Total Count: 24,394)			
Does not contain	Exact match	5,470	22.42%
Ends with type name	Ends with type name	4,625	18.96%
Does not contain	Ends with type name	3,447	14.13%
...other combinations		10,852	44.49%

for some other reason (in the 18.73% case). Next, we look at the data in this set of 18.73% to try and determine what these other reasons might be. While there was no visibly generalizable trend, we notice that rename instances in this set contain a mix of primitive and non-primitive data types associated with the original name of the identifier and, as part of the rename process, all primitive data types were converted to non-primitive data types (e.g., `long timestamp` → `Clock clock`). This might indicate that one of the trends for the primitives in this data is that these are a case of broaden-meaning changes, where identifiers with primitive types are made into objects with more data and behavior.

Summary for RQ4: Looking at the 53,962 instances of renames applied to both an identifier and its given type, 60% of these changes are Simple, while 39% are Complex. This contrasts with the general population of renames in our study (i.e., regardless of whether there was a change to the type), where 68% are Simple and 29% Complex (Table 5.2). Of the 16.31% of identifiers involved in this RQ, most added or preserved their type name during a rename refactoring. A minority removed their type name. We observe that renames which involve a change to the type name tended to also involve identifiers with names exactly matching their type. Whereas, when there was not a data type change with the rename, the type name was a substring and tended to be appended to the end of the corresponding identifier name. Generally, developers tended to add or keep type names during renames rather than remove them. More research is required to ascertain the degree to which type names negatively impact the identifier names that they are a part of, but

Table 5.13: Examples highlighting covariant and contravariant rename instances

Semantic Change Type	**Rename Instances**
	Covariant
Identifier Name: Narrow	`Mongo mongo` → `MongoClient mongoClient`
Data Type Name: Narrow	`Client client` → `ClientEditor clientEditor`
Identifier Name: Broaden	`TabComponent childTabComponent` → `Tab childTab`
Data Type Name: Broaden	`DateTime availabilityEnd` → `Duration availability`
Identifier Name: Preserve	`CsvCreator csvCreator` → `CsvMaker csvMaker`
Data Type Name: Preserve	`Log log` → `Logger logger`
	Contravariant
Identifier Name: Narrow	`SolrConfig solrConfig` → `String solrConfigFile`
Data Type Name: Broaden	`GraphRoute graphRoute` → `Object graphRouteObj`
Identifier Name: Broaden	`String fileName` → `File file`
Data Type Name: Narrow	`Executor workerPool` → `ExecutorService pool`
Identifier Name: Preserve	`String validationInformation` → `Message validationInfo`
Data Type Name: Narrow	`QueryOption reusable` → `QueryOptionReuse reuse`

it is possible to recommend developers reconsider whether there is a reason the type name *should* be part of the identifier during a rename. The trends in Table 5.12 are reported more fully in our openly available dataset.

5.3.6 RQ5: What semantic changes occur when an identifier and its corresponding type are changed together?

To answer this research question, we focus our analysis only on rename refactorings that included a change in data type (i.e., 53,962 or ≈ 17.39% of rename instances) and analyze how modifications applied to these names are reflected in their data type.

We examine how the semantic meaning of an identifier varies when there is a change to the associated data type. The majority of semantic updates involved a change in the meaning of the identifier. A drill-down into the change in meaning types shows that developers change the data type when Narrowing the meaning of the name approximately 67.91% of the time (e.g., `Parse parse` → `ParseResult parseResult`). A Broadening of the identifier names occurs 20.98% of the time (e.g., `String jobName` → `Job job`), followed by Preserve at 8.80% (e.g., `FormulaContext` `formula` → `ExpressionContext expression`), Add and Remove at 1.62%, and 0.64%, respectively. This contrasts somewhat with our findings on general renames (RQ3), because in RQ5 we find that these renames tend to narrow meaning more often (+23% more often), add meaning less often (-36% less often), and broaden meaning more often (+5%) compared to general rename semantic changes examined in RQ3. If we look at semantic updates made directly to the type name, approximately 69% (or 27,298) of the data type changes show a narrow in meaning, while 24% broadened with the remaining 7% belonging to add and remove.

We also look into how the semantic meaning of types and their corresponding identifier names covary. 71.94% (or 28,341) of the identifier and data type name changes show a covariant relationship; both the identifier and its associated data type name underwent the same semantic update. From a more granular view, we observe that the narrowing of an identifier and data type name occur the most, approximately 56.28% (or 22,171). An example of this type of occurrence is when the developer performs the following *Rename Attribute* operation: `DateFormat defaultDateFormat` → `DateTimeFormatter defaultDateTimeFormatter`. In this example, both the identifier name and data type undergo a narrowing of its respective original meaning. The next two highest occurrences were contravariant: a narrowing of the identifier name and broadening of the data type name at 12.64%; and covariant: a broadening of both the names at 11.02%. In Table 5.13, we provide examples of covariant and contravariant instances that occurred in our dataset.

Finally, we look at the relationship between identifier names being changed to/from plural form and their data type changing to/from a collection. To detect a change in plurality, we compare the matched terms in the old and new identifier names looking for either a singular to plural or plural to singular change between the matched terms. For example, when the developer renames the attribute `defaultValue` to `defaultValues`, the part of speech for the term 'Value' changes from singular to plural. At a high level, as shown in Table 5.14, the majority of renames did not undergo a data type change nor a change in the plurality of their name. However, if we were to focus on only instances that show a change in plurality, approximately 47.82% $(3407/(3407 + 3122))$ of plurality changes also had a change in data type (e.g., `List<String> contextNames` → `String contextName`), while the other 52.18% $(3122/(3407 + 3122))$ of plurality renames did not have a change of data type (e.g., `String hostName` → `String hostNames`).

We also detect when data types that were part of a rename were changed to or from a collection type. Table 5.15 provides a breakdown of the various combinations of single-reference and collection-based data types that underwent a change in data type. Our analysis shows that the majority of type changes (Table 5.15, $\approx$ 82.64%) were not group/collection based (i.e., neither the old or new

Table 5.14: Mapping between identifier name change in plurality and change in data type

Change in Data Type?	**Change in Plurality?**	**Count (Total: 310,309)**	**Percentage**
No	No	252,940	81.51%
Yes	No	50,840	16.38%
No	Yes	3,407	1.10%
Yes	Yes	3,122	1.01%

name utilized an array or collection-based data type). Identifiers that did utilize collection based data types in either the new, old, or both names (e.g., `List<String> contextNames` → `String contextName`) accounted for around 17.36%.

We use the data about plurality and data-type above to study how identifier name plurality and data-type are connected. We observe that around 69.47% of renames that did not have a change in plurality (but did have a type change; Table 5.16) also did not utilize collection-based data types in either the old or new name (e.g., `DateTime date` → `LocalDate day`). Additionally, around 3.74% (Table 5.16, $(900 + 1120)/53962$) of the instances whose data type changed to a collection-based data type change did not show a change in plurality. For example, even though the *Rename Attribute* refactoring: `String exportToolCommand` → `List<String> executableCommand` does not show an overall change in plurality, the developer performs a change in data type by moving from a non-collection to a collection based data type. When a data-type is modified such that it becomes a collection, 64.29% (Table 5.16, $1621/(900 + 1621)$) of the time there is a change in plurality for its corresponding identifier name and 35.7% of the time, there is no change in the plurality of the name. When a data-type is modified such that it ceases to be a collection, 53.02% (Table 5.16, $1264/(1264 + 1120)$) of the time there is a change in plurality for the corresponding name and 46.98% of the time, there is no change in plurality. One other interesting note about this table is that 13.17% of the time, when there was a change in type during a rename operation, the plurality of the identifier changed but we did not detect a collection type. This indicates that the objects' class may have internally changed to include some form of collection or collection-like behavior, which we would not be able to detect since we only look at type signatures without doing internal analysis on classes.

Summary for RQ5: From a semantic perspective, consistent with RQ3, we observe that developers generally narrow the name of the identifier in conjunction with a change in data type as opposed to other types of semantic change types. However, the data also shows that this frequency

Table 5.15: Distribution of data type changes with primitive/non-primitive and single/collection data types for rename instances that changed data type

Old Data Type	**New Data Type**	**Count**	**Percentage**
Primitive vs. Non-Primitive (Total Count: 53,862)			
Non-Primitive	Non-Primitive	49,380	91.51%
Primitive	Non-Primitive	2,532	4.69%
Non-Primitive	Primitive	1,157	2.14%
Primitive	Primitive	893	1.65%
Single vs. Collection (Total Count: 53,962)			
Single	Single	44,593	82.64%
Collection	Collection	4,464	8.27%
Single	Collection	2,521	4.67%
Collection	Single	2,384	4.42%

Table 5.16: Mapping between identifier name change in plurality and use of collection-based data type for rename instances that underwent a change in data type

Is Data Type a Collection?		**Change in**	**Count**	**%**
Old Identifier	**New Identifier**	**Plurality?**	**(Total: 53,962)**	
No	No	No	37,487	69.47%
No	No	Yes	7,106	13.17%
No	Yes	No	900	1.67%
No	Yes	Yes	1,621	3.00%
Yes	No	No	1,120	2.08%
Yes	No	Yes	1,264	2.34%
Yes	Yes	No	3,556	6.59%
Yes	Yes	Yes	908	1.68%

is more pronounced (i.e., higher) for renames which involve type changes. We also note that there was a decrease in *add meaning* changes and a slight increase in *broaden meaning* changes compared to the general set of renames from RQ3. Additionally, when a data-type is modified such that it becomes a collection, 64.29% of the time there is a change in plurality for its corresponding identifier name, and 35.7% of the time, there is no change in the plurality of the name. When a data-type is modified such that it ceases to be a collection, 53.02% of the time, there is a change in plurality for the corresponding name, and 46.98% of the time, there is no change in plurality. 1.68% of the time, the data type is already a collection object, and the identifier is modified to be plural to reflect this. Finally, we found that most identifier names covariantly evolve with their corresponding type name, and a minority of the renames we examined showed a contravariant relationship.

5.3.7 RQ6: What refactorings most frequently appear before and after an identifier and its corresponding type are changed together? Are there specific semantic changes which correlate with these refactorings?

To answer this question, we look at the refactorings that surround attribute, method, parameter, and variable rename refactorings that have a change in data type. Hence, the input data for this research question is a subset of the dataset used in RQ2; specifically the subset of renames which included a change in type. Except for class, we extract the subset of rename instances for attributes methods, and method variables that underwent a change in data type while being renamed. In total, 283 ($\approx$ 15.31%) attribute renames that underwent a data type change also had a refactoring occurring either before or after the rename. Similarly, we observe 564 ($\approx$ 9.63%) variable, and 734 ($\approx$ 6.78%) method renames under the same criteria.

Similar to RQ2, the majority of the refactorings occurred before a rename refactoring. Hence, we look at the refactorings that frequently occurred before a rename with a data type change. For variables, we observe that the majority of variable renames containing a data type change occurred approximately 42.73% of the time after the same variable was previously renamed. Rename-based data type changes for methods occurred 20.30% of the time after an *Extract Method* operation, and 16.89% of the time after a *Rename Variable* within the same method. This is nearly identical to RQ2 data (Table 5.5), where *Extract Method* and *Rename Variable* occurred 18.53% and 15.17% of the time, respectively, before a rename. Likewise, renames occurred after *Move Attribute* $\approx$ 66.08% of the time. This relationship is weaker than in Table 5.5, where renames occurred after *Move Attribute* 83.32% of the time.

Finally, we investigate the semantic updates made to the identifier's name, which follows a refactoring operation. Presented in Table 5.17, are the top three refactoring operations that preceded a rename refactoring that also had a data type change. This table also shows the distribution of semantic updates that the name undergoes. The trends mirror what we discussed in RQ4 and RQ5, but are broken down by refactoring which preceded the rename of an identifier name and its type. Add-meaning changes were much less likely when an identifier and its type are renamed together. If we compare Table 5.6 and Table 5.17, we can see that general identifier renames with a preceding *Move Attribute* refactoring tend to add meaning, but when we narrow to identifier renames which change the type in-tandem, we see a sharp decline in the relative number of add-meaning changes (a reduction from 54% to 1.07%) and instead see a majority of narrow and broadening-meaning changes. A similar drop occurs for identifier renames with a preceding *Rename Variable* (from 77% to <3%). This data breaks down some of the trends we note in RQ5; showing us that, for example, the loss of add-meaning changes has some context (i.e., *Move Attribute* when an identifier and its type are renamed) which may be leveraged when understanding, or recommending/suggesting, renames.

Summary for RQ6: Comparing the refactoring co-occurrence data from RQ2 with RQ6, our findings from RQ6 are similar to our RQ2 findings in that the refactorings occurring before the rename are more or less the same (i.e., *Rename Variable*, *Move Attribute*, and *Extract Method*). However, we also find that the relationships with these refactorings in RQ6 are generally weaker or roughly the same as in RQ2. This indicates that a rename in which a data type is changed may be less likely to have a co-occurring refactoring. In RQ5, we found that narrow- and broaden-meaning changes are emphasized while add-meaning is de-emphasized when an identifier and its type change together versus general renames. In RQ6, we further broke this trend down and see that the reduction, while pervasive, heavily affects refactoring contexts, as we can see if we compare semantic changes made to renames correlated with *Move Attribute* in Table 5.6 with the same in Table 5.17 and note the significant drop in add-meaning changes (a reduction from 54% to 1.07%). This data indicates that renames which include type changes may need to be treated as special cases in any future rename recommendation/analysis effort due to the relationship between the identifier and its corresponding type.

Table 5.17: An overview of the types of semantic updates an identifier name with a data type change undergoes when preceded by another refactoring operation

Identifier Type	**Refactoring Before Rename**	**Top 3 Types of Rename Forms**	**Type of Semantic Update**	**Top 3 Semantic Change Subtypes**
Attribute	Move Attribute (Total Count: 187)	Simple (65.57%) Complex (36.90) Formatting (0.54%)	Change (90.37%) Preserve (9.63%)	Narrow (77.01%) Broaden (41.71%) Add (1.07%)
	Pull Up Attribute (Total Count: 61)	Simple (54.1%) Complex (44.26%) Formatting (1.64%)	Change (91.80%) Preserve (8.20%)	Narrow (54.46%) Broaden (26.23%) Remove(6.56%)
	Push Down Attribute (Total Count: 16)	Simple (56.25%) Complex (43.75%)	Change (87.5%) Preserve (12.5%)	Narrow (68.75%) Broaden (18.75%)
Method	Extract Method (Total Count: 149)	Simple (56.38%) Complex (43%) Formatting(0.67 %)	Change (91.28%) Preserve (8.72%)	Narrow (62.42%) Broaden (24.83) Add(0.67%)
	Rename Variable (Total Count: 124)	Complex (50.81%) Simple (47.58%) Formatting(0.81%)	Change (91.13%) Preserve (8.87%)	Narrow (66.94%) Broaden (21.77%)
	Rename Method (Total Count: 105)	Simple (67.62%) Complex (32.38%)	Change (83.81%) Preserve (16.19%)	Narrow (45.71%) Broaden (34.29%) Add(2.86%)
Variable	Rename Variable (Total Count: 241)	Simple (56.85%) Complex (43.15%)	Change (97.1%) Preserve (2.9%)	Narrow (59.75%) Broaden (32.37%) Remove (3.73%)
	Extract Variable (Total Count: 95)	Simple (64.2%1) Complex (33.68%) Formatting (2.11%)	Change (90.53%) Preserve (9.47%)	Narrow (70.53%) Broaden (19.95%) Remove (1.05%)
	Replace Variable With Attribute (Total Count: 3)	Complex (66.67%) Simple (33.33%)	Change (100%)	Narrow (66.67%) Broaden (33.33%)

Chapter 6

On the generation, structure, and semantics of grammar patterns in source code identifiers

The contents of this chapter are part of the study "***On the generation, structure, and semantics of grammar patterns in source code identifiers***" published in the Journal of Systems and Software [177].

6.1 Introduction

Currently, developers spend a significant amount of time comprehending code [121, 165]; 10 times the amount they spend writing it by some estimates [165]. Studying comprehension will lead to ways that not only increase the ability of developers to be productive, but also increase the accessibility of software development (e.g., by supporting programmers that prefer top-down or bottom-up comprehension styles [132, 221]) as a field. One of the primary ways a developer comprehends code is by reading identifier names which make up, on average, about 70% of the characters found in a body of code [125].

Recent studies show how identifier names impact comprehension [118, 144, 155, 206, 213], others show that normalizing identifier names helps both developers and research tools which leverage identifiers [107, 176]. Thus, many research projects try to improve identifier naming practices. For example, prior research predicts words that should appear in an identifier name [84, 89, 152, 161]; combines Natural Language (NL) and static analysis to analyze or suggest changes to identifiers [84, 101, 103, 106, 137, 140, 145, 185, 191]; and groups identifier names by static role in the code [94, 127, 175].

One challenge to studying identifiers is the difficulty in understanding how to map the meaning of natural language phrases to the behavior of the code. For example, when a developer names a method, the name should reflect the behavior of the method such that another developer can understand what the method does without the need to read the method body instruction set. Understanding this connection between name and behavior presents challenges for humans and tools; both of which use this relationship to comprehend, generate, or critique code. A second challenge lies in the natural language analysis techniques themselves, many of which are not trained to be applied to software [107], which introduces significant threats [150]. Addressing these problems is vital to improving the developer experience and augmenting tools which leverage natural language.

One of the most popular methods for measuring the natural language semantics of identifier names is part-of-speech (POS) tagging. While some work has been done studying part-of-speech tag sequences (i.e., grammar patterns) in identifier names [113,115,137,139,145,157], these prior works either focus on a specific type of identifier, typically method or class names; or discuss grammar patterns at a conceptual level without showing concrete examples of what they look like in the wild, where they can be messy, incomplete, ambiguous, or provide unique insights about how developers express themselves. In this study, we begin addressing these issues. We create a dataset of 1,335 manually-annotated (i.e., POS tagged) identifiers across five identifier categories: class, function, declaration-statement (i.e., global or function-local variables), parameter, and attribute names. We use this dataset to study and show concrete grammar patterns as they are found in their natural environments.

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

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

RQ2: How accurately do the chosen taggers annotate grammar patterns and individual tags? This question explores the accuracy of the part-of-speech tagger annotations versus human annotations. We determine which patterns and part-of-speech tags are most often incorrectly

Table 6.1: Total number per category of identifiers and unique grammar patterns across all systems

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

generated by tools.

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

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

6.2 Methodology

Identifiers come in many forms. The most common fall into one of the five following categories: class names, function names, parameter names, attribute names (i.e., data members), and declaration-statement names. A declaration-statement name is a name belonging to a function-local or global variable. We use this terminology as it is consistent with srcML's terminology [119] for these variables and we used srcML to collect identifiers. Therefore, to study grammar patterns, we group a set of identifiers based on which of these five categories they fell into. The purpose of doing this is two-fold. 1) We can study grammar pattern frequencies based on their high-level semantic role (i.e., class names have a different role than function names). 2) We can consider the differences in accuracy for part-of-speech taggers when given identifiers from different categories.

6.2.1 Setup for the dataset of human-annotated identifiers

We created a gold set of grammar patterns for each of the five categories above by manually assigning (i.e., annotating) part-of-speech tags to each word within each identifier within each of the five categories above. We calculate the sample size by counting the total number of identifiers in each of the five categories (given in Table 6.1) and calculating a sample based on a 95% confidence level and a confidence interval of 6%. We chose this confidence level and interval as a trade-off between time (i.e., annotating and validating is a manual effort) and accuracy. Using this confidence level and interval, we determine that each of our five categories needs to contain 267 samples (i.e., 267 was the largest number any of the sets required to be statistically representative, some required less– we went with 267). This totals to 1335 identifiers in the entire set. This sample size is also similar to the number used in prior studies on part-of-speech tagging [137, 180].

Initially, one annotator was assigned to each category and was responsible for assigning grammar patterns for each of the 267 identifiers in the category. The annotators were given a split identifier (using Spiral [146]) along with the identifier's type and, if the identifier represented a function, the parameters/return types for that function. They were also allowed to look at the source code from which the identifier originated if needed. The annotators were asked to additionally identify and correct mistakes made by Spiral.

We did not expand abbreviations. The reason for this is that abbreviation expansion techniques are not widely available (e.g., cannot be easily integrated into different languages or frameworks, cannot be readily trained, are not fully or publicly implemented) and still not very accurate [176]. Therefore, a realistic worst-case scenario for developers and researchers is that no abbreviation-expansion technique available to use; their part-of-speech taggers must work in this worst-case scenario. We also tried not to split domain-term abbreviations (e.g., Spiral will make IPV4 into IPV 4; we corrected this back to IPV4). We did this because some taggers may recognize these domain terms. Furthermore, we are also of the view that these terms should be recognized and appropriately tagged in their abbreviated (i.e., their most common) form. In the future, we plan to train a part-of-speech tagger using this dataset.

After completing their individual sets, we traded and reviewed one another's sets (i.e., performed cross-validation) twice. Thus, every identifier has been reviewed by two annotators. There was only one disagreement that could not be settled due to a particularly disfigured identifier; therefore, one identifier was randomly re-selected. This identifier is as follows: *uint8x16_t a_p1_q1_2*; the annotators could not ascertain the meaning of the letters and numbers, making it difficult to tag. Once every identifier was assigned a grammar pattern manually and had been reviewed by at least two other annotators, we ran each of our three part-of-speech taggers on the set of split identifiers; providing whatever information was required by the tagger (e.g., some taggers require full function signature, others only use the identifier name). We used srcML [119] to obtain any additional information required by the taggers. The grammar pattern output from each tagger was used to generate frequency counts and compare to the manually-annotated grammar patterns to calculate accuracy.

6.2.2 Definition of Accuracy

Accuracy in this study is synonymous with agreement; we compare the automatically generated annotations from the individual part-of-speech taggers with the manual annotations provided by humans. To be specific, we perform two different accuracy calculations for each tagger. One to determine the accuracy of each tagger on the individual part-of-speech tags found in Table 6.7 and one to determine the accuracy of each tagger on full grammar patterns like those in Table 6.2. To put this into an equation, we first define four sets. H_{gp}, the set of all human-annotated grammar patterns. T_{gp}, the set of all tool-annotated grammar patterns for a single part-of-speech tagger; there are three of these since we use three tools in this study. H_{word}, the set of all human annotations for individual words in our set. Finally, T_{word}, the set of all tool annotations for individual words. Again, there are three of these since we use three part-of-speech tools in this study. We then define grammar pattern level accuracy as the number of patterns which the human and tool sets agreed on (i.e., intersection) divided by the total number of grammar patterns annotated by humans. The equation follows:

$$\left|H_{gp} \cap T_{gp}\right| \div \left|H_{gp}\right| \tag{6.1}$$

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

$$\left|H_{word} \cap T_{word}\right| \div \left|H_{word}\right| \tag{6.2}$$

Table 6.2: Top 5 patterns in dataset along with frequency of each pattern and % of the set represented by that pattern

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

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

6.2.3 Data Collection

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

We provide the list of systems and their characteristics in Table 6.3. The systems we picked were 615 KLOC on average with a median of 476 KLOC, a min of 30 KLOC, and a max of 1,800 KLOC. Further, most of these systems have been in active development for the past ten years or more and all of them for five years or more. The younger systems in our set are popular, modern programs. For example, Swift is a well-known programming language supported by Apple, Telegram is a

Table 6.3: Systems used to create dataset

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

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

To collect the 1,335 identifiers, we scanned each of our 20 systems using srcML [119] and collected both identifier names/types and the category that they fell into (e.g., class, function). Then, for each category, we randomly selected one identifier from each system using a round-robin algorithm (i.e., we picked a random identifier from system 1, then randomly selected an identifier from system 2, etc. until we hit 267). This ensured that we got either 13 or 14 identifiers from each system (267/20 = 13.35) per category and mitigates the threat of differing system size.

6.3 Experimental Results

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

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

Table 6.2 contains data from the human-annotated set separated into categories based on the location of the identifier in code. The table shows the top five grammar patterns for each category and each of the three taggers we used to generate the grammar patterns. In addition, Figure 6.1

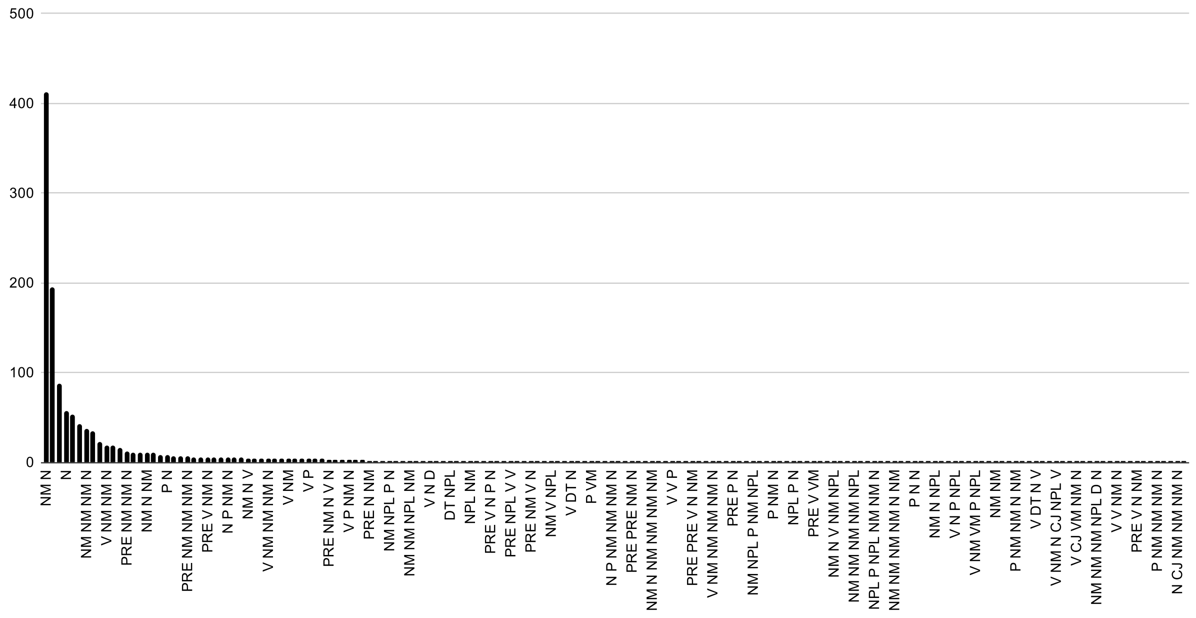


Figure 6.1: Distribution of unique grammar pattern frequency over entire dataset – not all unique patterns are shown due to space.

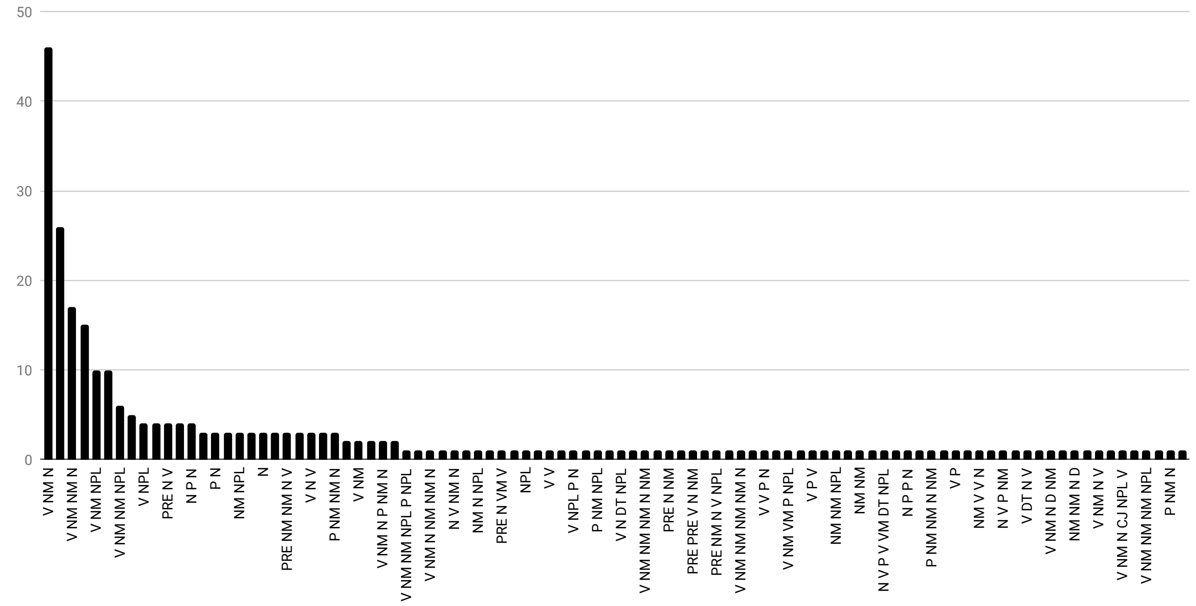


Figure 6.2: Distribution of unique method grammar pattern frequency – not all unique patterns are shown due to space.

Table 6.4: Grammar patterns from which other grammar patterns are frequently derived.

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

shows the distribution of unique grammar patterns across all identifiers. It shows that a minority of the total grammar patterns found in the set are repeated frequently, while the majority occur only once. Due to the fact that functions had the highest number of unique grammar patterns (Table 6.1), we also show the distribution of unique function grammar patterns in Figure 6.2. The distribution is largely similar to the prior figure with all unique grammar patterns, but the most common function grammar patterns are different than the general set due to the semantics being conveyed by function names (e.g., use of verbs to convey actions) versus other identifiers. There are three grammar patterns from which most frequent grammar patterns we will discuss are derived. These are shown and described in Table 6.4. Specifically, they are the verb phrase, noun phrase, and plural noun phrase patterns. We now discuss the most common grammar patterns found in our data set.

Grammar Pattern *PRE* NM+ N*: Looking at Table 6.2, the *NM N* instance of this pattern appears in the top five most frequent in each category, and it is the most ubiquitous pattern in our dataset. Noun modifiers are used to modify developers' understanding of the entity referred to by the head-noun (Table 6.4). **Examples**: *factory class*, *auth result*, and *previous caption.* These identifiers embody the noun phrase concept. The specific entities that they reference are *class*, *result*, and *caption* respectively. While the noun modifiers (factory, auth, and previous) specify some characteristics of the entity they comes before. The *class* is not just any kind of class; it is a *factory* class, the *result* is specifically the consequence of some form of *authentication*, the *caption* being referred to is the *previous* in relation to some other caption. Adding more noun modifiers emphasizes this effect. **Examples:** *max buffer size* uses *max* and *buffer* to describe characteristics

Table 6.5: Frequency at which identifiers with a verb in their grammar pattern also have a boolean type and frequency at which identifiers with a boolean type have a verb in their grammar pattern

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

of the *size*, which is the head-noun. The same applies to *previous initial value* and *network security policy.*

This pattern is found in all categories, but it is somewhat out of place in the Function Name category, since function names tend to contain a verb. We manually investigated these instances and found that many of them are functions with an implied verb. **Examples:** *deep Stub.* It contains no verb, but its behavior is to return (i.e., *get*) a *deep stub* object. Another example is the *lower Boundary Factor* function which returns the *lower boundary factor* based on a parameter. Many of these functions are getters or setters where the *get* or *set* is not in the name of the function.

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

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

Grammar Pattern *V NM+ N*: This pattern and its extensions are most common in functions. This is a verb phrase pattern, where a verb is followed by a noun phrase (Table 6.4). **Examples**: *check gl support*, *resize nearest neighbor*, and *get max shingle diff.* In *check gl Support*, the noun phrase specifies what we are interested in (openGL support), *check* tells us that we are checking a condition– specifically that *gl support* is available. The same applies for the other two identifiers; specifying which neighbor to *resize* in *resize nearest neighbor* and the nature of the *diff[erence]* to return (i.e., *get*) in *get max shingle diff.*

One characteristic we observed about this and other verb-based patterns is that, when it appears

Table 6.6: Frequency at which identifiers ending with a plural also have a collection type and frequency at which identifiers with a collection type end with a plural

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

outside of function names, it tends to be for an identifier with a boolean type or a type which can be treated as boolean (e.g., integer). **Examples**: *add bias to embedding*, *is first frame*, and *will return last parameter*. This is because boolean variables act like predicates; asking a question whose answer is true or false (e.g., add bias to [the] embedding?, is [this] the first frame?, will [this] return [the] last parameter?). Other researchers have made this observation [106, 137], but only one reports quantity [113]. In Table 6.5, we show two things: 1) the percentage of all identifiers in the dataset with a at least one verb in their grammar pattern and a type which can be interpreted as boolean. 2) the percentage of all identifiers in the dataset with a boolean type that also have a at least one verb in their grammar pattern.

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

Grammar Pattern *V* NM+ NPL*: This pattern has two configurations. It is either a plural noun phrase pattern or a plural verb phrase pattern (Table 6.4). Assuming that the use of a plural must be significant somehow, since some sources advise using plural identifiers for collections and certain types of classes [96], we analyzed our dataset to see if identifiers which have a plural head-noun were more likely to have a type which indicated a collection (e.g., list, array) type. To do this, we examine the type name for each identifier and record if it contains the words: *list, map, dictionary, collection, array, vector, data, or set.* We additionally record if the type name is plural (e.g., ending in -s) or if the identifier has square brackets (i.e., []) next to it. We then manually check these to see if they were really collection types. The results of this investigation are in Table 6.6. This table shows two perspectives on the data: 1) how many identifiers with a collection type also have a plural head-noun. 2) how many identifiers that have a plural head-noun also have a collection type.

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

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

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

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

Summary for RQ1: Table 6.2 contains the most frequent grammar patterns in the human-annotated set. We identified five patterns by looking at how frequently they occurred in our human-annotated dataset. By far, the most ubiquitous pattern we found was the noun phrase (*NM+ N*) pattern as it appears in every category. We also found that verb phrase (*V NM+ N)* patterns are most common in function names but also appear in other types of identifiers; specifically those with a boolean type, and that plural noun phrases (*NM+ NPL*) have a somewhat heightened chance of representing collection identifiers. Results indicate that 1) due to the ubiquity of noun phrase patterns, part-of-speech taggers should predict *NM* on unknown, non-numeric tokens that are not the right-most token in the identifier; *N* is a better prediction for the right-most token, as it is likely the head-noun. 2) our data supports the observed, heightened appearance of verbs in boolean variables from prior work. 3) while there is a link between identifier names containing a plural and collection data types, the plural is sometimes used to reference multiplicity of related, heterogeneous data (i.e., class member data), particularly for function identifiers. This presents an opportunity to support developers in how, and when, to use plurals. The grammar patterns we generated highlight how the name of an identifier is influenced by the semantics of the language and gives us a glimpse into how developers use words in an identifier to comprehend their code.

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

We compare the output of the three part-of-speech taggers in this study with the manually-annotated grammar patterns in order to calculate the accuracy of each tagger at both the level of grammar patterns and the level of individual part-of-speech tags. Please refer to Section 6.2.2 for an explanation of how we calculate accuracy. Starting with Table 6.7, which contains our per-tag (i.e., word-level) accuracy analysis, we observe that Swum had the highest agreement with the human annotations with respect to noun modifiers, determiners, and preambles. Stanford had the highest agreement with respect to everything else. Posse never outperformed both the other two in a single category, but did perform better than Stanford at annotating noun modifiers and better than Swum at annotating nouns, prepositions, and verbs. The numbers here indicate that Stanford has the best all-around performance when we are looking at accuracy on individual part-of-speech

Table 6.7: Frequency of per-tag agreement between human annotations and tool annotations

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

Table 6.8: Percentage of tool-annotated grammar patterns which fully match human-annotated grammar patterns

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

annotations, as it is able to detect a broader range of them more accurately than either Swum or Posse. However, while Stanford had the highest accuracy on the largest number of part-of-speech tag types, Swum tagged the highest number of raw words correctly with 2,703 correct annotations versus Stanford's 1,854. The results indicate areas of strength for each tagger; their combined output may increase their overall accuracy.

With this context in mind, we will now look at the agreement between the tagger-annotated and human-annotated grammar patterns (i.e., identifier-level accuracy analysis). This is shown in Table 6.8. We broke Stanford down into several columns in this table to see how its accuracy changes when we configure it differently. The configurations are as follows: We add an *I* before function names before applying Stanford tagger. Additionally, some types of verbs can be considered adjectives in different contexts. Thus, we test the accuracy of Stanford under either assumption; the verb being used as a verb or being used as an adjective.

This data shows that Swum had the highest agreement with the human-provided annotations at the level of grammar patterns. Stanford had the second-highest agreement on average when we assume its best configuration– though, Posse has a higher agreement with the human annotations in the Classes category regardless of Stanford's configuration. Swum's accuracy ranged between 50.2% and 67.4% while Posse and Stanford's ranged between 13.1% - 24.7% and 9.7% - 26.6% respectively. The difference in the results between Tables 6.7 and 6.8 are interesting in that Stanford has the best performance in Table 6.7 but under-performs Swum by a large margin in Table 6.8. The reason for this difference is the ubiquity of noun modifiers in identifier names. Even though Stanford is more accurate on a larger set of part-of-speech tag categories, it under-performs on noun modifiers compared to Swum (15.71% accuracy for Stanford vs 94.01% for Swum), which consistently annotates noun modifiers correctly. Noun modifier is the most frequent annotation (with a frequency of 1,604 per Table 6.7); Swum gets 1508 of these correct while Stanford gets 252, meaning Stanford missed 1256 words that Swum got correct. If we combine this with the fact that Swum's performance on the second-most-common annotation, nouns, is much closer to Stanford's (85.54% accuracy for Swum vs 93.25% for Stanford) than Stanford's performance is to Swum's on noun modifiers, it makes sense that, when looking at accuracy on annotating full identifier name grammar patterns (i.e., Table 6.8), Swum outperforms Stanford despite Stanford's high annotation accuracy on most other part-of-speech tag types. In short, Stanford is more likely to get the very common *NM+ N* pattern incorrect due to mis-annotating NM compared to Swum, which will occasionally mis-annotate N, but not as frequently as Stanford will mis-annotate NM.

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

To understand more about the differences in agreement between the humans and taggers, we identified which patterns were most frequently incorrectly generated for each tagger, in part to understand each tagger's weaknesses. These patterns represent paths toward significantly improving the accuracy of part-of-speech taggers for source code identifiers. Table 6.9 gives the top five most frequently mis-annotated grammar patterns per category for each tagger used in our study.

To contextualize the details we discuss below, we quickly summarize some of the core problems found in the part-of-speech tagger output using the data in Table 6.9. Posse has trouble generating noun phrase and verb phrase patterns (i.e., *NM+ N* and *V NM+ N* respectively), including plural noun phrases such as *NM+ NPL*. The reason for this is that Posse does not generally identify noun modifiers in sequences greater than 1 (i.e., it may annotate *NM N*, but never *NM NM N*). Even on sequences with only one noun modifier, it tends to prefer annotating noun modifiers as a noun. This problem is more pronounced in Stanford, which generally missed more noun phrase and verb phrase patterns than Posse. This makes sense as Stanford did not annotate noun modifiers very well (Table 6.7). However, Stanford is very good at identifying plurals; Swum and Posse never identified plural words correctly in the dataset (Table 6.7). This is why plural noun and verb phrase patterns are both frequently mis-annotated by both Swum and Posse. We will now focus our discussion around specific, commonly mis-annotated grammar patterns and discuss tagger annotations in the context of each.

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

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

Grammar Pattern *NM* NPL*: Posse and Swum do not detect plurals, while Stanford is very good at detecting them (see Table 6.7). Stanford tends to get noun plurals individually correct even when it would mistakenly annotate a noun modifier as a noun, which is why *NM NPL* was still one of the most common patterns for Stanford to mis-annotate. **Example**: *num active contexts* has a pattern *NM NM NPL* due to the plural at the end. Swum does not recognize the plural and gives it a *NM NM N* pattern. Posse gave it an *N NM N* pattern and Stanford gave it a *N NM NPL* pattern. Thus, Stanford and Swum were both nearest to the correct solution despite both being

Table 6.9: Top 5 Most frequently mis-annotated grammar patterns

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

incorrect. Stanford did not have trouble with *NPL* patterns when there were no noun modifiers. This suggests a very good way that tagger annotations may be combined; Stanford can identify noun plurals accurately and Swum can identify noun-modifiers accurately. In general, this was the hardest, frequently observed pattern for any individual tagger to annotate completely correctly.

Grammar Pattern *V NM+ N*: The deciding factor in mis-annotating verb phrase patterns tended to be noun modifiers and plural nouns. Generally speaking, the taggers agreed with the human annotators on the position of the verb in method names between 56% and 76% of the time. This contrasts with how often they agreed on the full human annotation in function names (Table 6.8); 24.7% of the time for Posse, 50.2% of the time for Swum, 25.5% of the time for Stanford. All taggers still have problems determining the correct verb, but detecting noun modifiers is the bigger issue.

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

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

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

Grammar Pattern *PRE...*: We will discuss all patterns with a preamble here (i.e., the ... could be any other pattern, such as noun phrase or verb phrase). Preambles were difficult for all taggers to deal with. Swum rarely detects preambles while Posse and Stanford do not detect them at all. Generally, a preamble is mis-annotated as noun for Posse and Stanford or noun modifier for Swum. The problem with detecting preambles is that some prior information is required– a tagger needs to scan the code and/or be able to recognize naming conventions such as Hungarian [209], to determine which character sequences are being used as preambles. There are also cases where it is not clear that a frequent character sequence is a preamble. **Examples**: the GRPC project tends to append *grpc* before many of its identifiers– *grpc json writer value string* has a grammar pattern of *PRE NM NM NM N*. A scan of GRPCs code could identify this as a preamble. *XML* is sometimes used frequently at the beginning of function names such as *xmlWriter, xmlReader*. It may look like a preamble in some systems due to how frequently it appears at the beginning of an identifier, but is not a preamble because it specializes our understanding of the words *reader* and *writer*. This suggests that some domain knowledge is additionally required to correctly determine when a character sequence is a preamble.

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

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

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

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

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

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

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

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

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

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

Grammar Pattern *V P NM N:* This pattern is a verb followed by a prepositional phrase. This was found only among function identifier names. **Examples**: *convert to php namespace* and *register with volatility spread.* This pattern uses a verb to specify an action on an unknown entity (e.g., the identifiers above do not specify what to *convert* or what to *register*) and uses the noun phrase

Table 6.10: Grammar patterns broken down by language

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

on the right side of the preposition as a reference point; the specific thing to which this unknown entity will be related. The nature of this relationship is specified by the verb and preposition (i.e., convert to, register with).

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

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

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

Programming languages have their own individual characteristics. To give a few (i.e., non-exhaustive) examples: C is a procedural language that does not support objected oriented programming, Java is an object oriented language, and C++ supports facets of object orientated programming and generic programming. Given this, we grouped grammar patterns in our data set by programming language with a goal varying the language to see if certain grammar patterns were more common to a specific language. This data is shown in Table 6.10, where we show the top 10 patterns C, C++, and Java. We note that most of the identifiers in our set were either C++ (573) or Java (511) identifiers; a smaller number were C (229) and C-sharp (22). We leave C-sharp out of our analysis due to the very small number of them. We include C in our analysis, but note that the results for C may not generalize as well as for C++ and Java.

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

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

We then looked at the distribution of abbreviations and dictionary words between Java and C/C++ to provide more insight about the differences in identifier structure between languages. Since

Table 6.11: Distribution of abbreviations and dictionary terms between different languages in the data set

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

Table 6.12: Accuracy of taggers on abbreviated and non-abbreviated terms

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

abbreviations may make it difficult to obtain the meaning of a word, part-of-speech taggers might annotate these words less accurately. Table 6.11 shows the distribution. We include C# in this table for completeness despite its low number of identifiers. To determine if a token is a full word or an abbreviation, we used Wordnet [168]. If Wordnet recognized the word, we considered it a full word. Otherwise, it is an abbreviation. The results indicate that C and C++ tend to contain more abbreviated terms.

Given this, we then looked at the accuracy of each part-of-speech tagger on identifiers from different languages. This data is found in Table 6.12. All taggers had decreased performance on abbreviated terms, but Posse was the least accurate and saw the most significant decrease in performance (-14% from its full word accuracy). Finally, given this data about tagger performance on abbreviations and the distribution of abbreviations in different languages, we looked at tagger accuracy per programming language. This data is in Table 6.13. While Posse/Stanford performed better on Java than C/C++ systems, their performance degraded a total of 3.4% and 1.4% respectively. Swum, however, was nearly 12% less accurate on C identifiers compared to C++ identifiers. Swum also performed better on C++ identifiers versus Java, unlike the other two taggers.

Summary for RQ4: At the level of grammar patterns, while only controlling for identifier category (e.g., function name) and programming language, there does not appear to be a significant difference in the grammar patterns for Java and C/C++ identifiers. It may be the case that significant grammar pattern differences appear when controlling for more confounding factors and

Table 6.13: Accuracy of part-of-speech taggers split by programming language

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

we believe this would be a strong direction for future work. We do note a difference in the use of abbreviations between C/C++ and Java identifiers, where Java identifiers tend to have fewer abbreviations than the former two languages. Abbreviations can hinder the accuracy of part-of-speech taggers, which we confirmed by examining the annotations given to abbreviations by the three taggers in this study; all taggers performed worse on abbreviations than on full words. However, difficulty with abbreviations did not significantly reduce Posse/Stanford's performance between programming languages, indicating that abbreviations are not the biggest problem these taggers face, while expanding abbreviations may significantly (up to 10%) improve Swum's performance.

Chapter 7

Using Grammar Patterns to Interpret Test Method Name Evolution

The contents of this chapter are part of the study "***Using Grammar Patterns to Interpret Test Method Name Evolution***" published in the Proceedings of the 29th International Conference on Program Comprehension [190].

7.1 Introduction

In software, test methods names are constructed to describe both the entity that is being tested as well as actions taken by the test [225]. The name of a test is important for the same reason production method names are important; they help developers understand the purpose of the method. Further, these names can be used by automated approaches to analyze/understand test methods and automatically generate code for the test methods. Prior research indicates that test method names have a different structure than production method names [222, 225], but understanding how they are similar or different is still a problem that has not been sufficiently addressed. Prior studies on method naming focus on detecting linguistic anti-patterns [102], method naming bugs [145], and a multitude of naming/renaming practices [101, 161, 177, 182, 185, 191], but do not differentiate between production and test method naming structures. For this reason, the concepts discussed in these papers may not fully generalize to test method names. This will hinder our ability to improve and support test method name quality both in the case where they are manually written by developers or automatically generated by tools. It is important to consider the unique structure of test method names to complement and increase the impact of prior work by taking into account the unique structure and purpose of test method names.

We begin addressing this problem by studying the evolution of method name structure and semantics in test suites by, primarily, analyzing the sequence of part-of-speech (POS) tags, called grammar patterns [177], associated with the method's name. The purpose of this type of analysis is to understand how the semantics behind the test method name change and how these semantics correlate with changes to the actual testing behavior, as defined in the code. POS tags are obtained by splitting an identifier name into its constituent words and then annotating the split identifier manually. A grammar pattern provides us with a template-like sequence of POS tags, which are an abstract representation of an identifier's meaning.

One problem with analyzing identifier names is that it is difficult to automatically determine the meaning of words in an identifier and how these words interact with one another. It is even more challenging to take this meaning and use it to understand how it influences, or is influenced by, the behavior of the code. Grammar patterns allow us to perform this analysis more efficiently by broadly categorizing words into their corresponding POS; this allows us to relate words together and, also, we can relate different POS tags with certain types of code behavior [177]. *The goal of this study is to understand how test method names are structured, how they evolve in structure and meaning, and how the structure/meaning of these names relate to statically-verifiable code behavior. The data obtained in this study will be used to facilitate test name recommendation and appraisal.* We answer the following research questions:

RQ1: Based on the grammar patterns, how are test methods typically structured, how does this structure evolve, and how does it compare to previously-defined naming patterns? This question helps us understand the common grammar patterns latent in test method names, how they change over time, and how they are related to the code's behavior. We use this data to 1) understand the relationship between code behavior and grammar patterns, 2) compare our findings with prior work that taxonomizes test names at a coarser level; allowing us to determine whether our finer-grain analysis creates more and/or different patterns, and 3) compare against production name grammar patterns to help us pinpoint the differences between test and production naming structures.

RQ2: How are changes to the grammar pattern related to changes in the semantic meaning of the corresponding method name? Grammar patterns provide a way for us to learn the relationship between words, which is more granular than prior approaches, without comparing their specific definitions. This question explores how changes to the grammar pattern relate to changes in the meaning of a method name using a taxonomy, first defined by Arnaoudova et al. [101].

RQ3: What are the most common term changes, and what is the relationship between

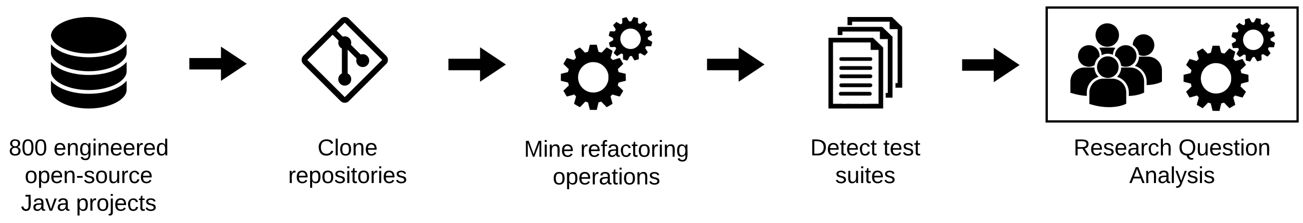

800 engineered open-source Java projects
Clone repositories
Mine refactoring operations
Detect test suites
Research Question Analysis

Figure 7.1: Overview of our experiment design.

the added term and removed term? In this research question, we look at the most frequent, concrete changes to words when a test method is renamed without using grammar patterns. The purpose of this is to give us an understanding of how these concrete changes are related to the changes we identify when using grammar patterns and to provide us more information about how these concrete names, and their evolution, relate to code behavior.

7.2 Methodology

Depicted in Figure 7.1 is an overview of our experiment design. We explain, in detail, each activity of our study in the subsequent subsections.

Projects: The projects in our study consist of 800 open-source Java projects hosted on GitHub. These projects belong to a curated dataset of engineered software projects, synthesized by the Reaper tool [172]. The projects in this dataset utilize software engineering practices such as documentation, testing, and project management. Not only do we clone each repository, but we also extract commit-level metadata by enumerating over the commit log of each project. The metadata we extract includes the timestamp of the commit, the author of the commit, and the files associated with each commit.

Refactorings: We utilize RefactoringMiner [218] for mining the rename refactoring operations from each project in our dataset. RefactoringMiner iterates over the entire commit history of a project in chronological order and compares the changes made to Java source code files in order to detect refactorings. RefactoringMiner is a state-of-the-art tool with a precision of 98% and a recall of 87% [208, 219]. Furthermore, we conduct our experiments on the entire commit history of the project (and not on a release-by-release comparison).

Test Suites: To identify test suites in the projects, we follow an approach similar to [187]. We first extract all Java source files (i.e., files with extension '.java') that underwent a refactoring. In this study, we focus on projects utilizing the JUnit testing framework [34]. Next, using JavaParser [33], we parse the Java files by building an abstract syntax tree for each source file. We mark a file as a unit test file if the file contains JUnit import statements (i.e., `org.junit.*` or `junit.*`) and a test method. For a file to contain a unit test method, the method should have an annotation called `@Test` (JUnit 4), or the method name should start with 'test' (JUnit 3). In total, we detected 319,108 unit test files, out of which only 12,010 test files had undergone a *Rename Method* refactoring.

Research Question Analysis: For our research questions, we make use of the data mined/extracted by the prior activities. The activities involved in answering each research question involve a combination of manual analysis (including data annotation) and quantitative analysis using custom-built tools/scripts. We detail our activities when addressing each research question when reporting our results.

7.3 Experimental Results

In this section, we report on the findings of our experiments.

7.3.1 RQ1: Based on the grammar patterns, how are test methods typically structured, how does this structure evolve, and how does it compare to previously-defined naming patterns?

In this RQ, we examine the POS associated with the old and new names to identify grammar patterns that are specific to test methods. We answer this research question with three sub-RQs. The first sub-RQ looks at the frequently occurring grammar patterns that occur in test method names, while the second sub-RQ compares common grammar prefix patterns reported by prior studies, on test ([222]) and production ([177]) method names, with findings from our dataset. Finally, the third sub-RQ examines how the POS changes when the method is renamed. The goal of this RQ, and therefore the sub-RQs, is to identify patterns in the way test method identifiers and their grammar patterns evolve through renames to understand how we can take advantage of this evolution in future research to provide developers with useful feedback about their renaming practices.

Approach: To understand the POS tags that constitute a method name in a unit test file, we manually annotated a statistically significant sample of renamed methods. In total, 632 test method rename instances (i.e., the old and new names) were manually analyzed. The analyzed sample represents the original set of 12,010 renamed methods, with a 99% confidence level and a 5%

confidence interval. Similar to prior research [177], our annotation process considered 10 English POS tags– noun (N), determiner (DT), conjunction (CJ), preposition (P), noun plural (NPL), noun modifier/adjectives (NM), verb (V), verb modifier/adverbs (VM), pronoun (PR), and digit (D). Our annotation process consisted of three stages– the annotation stage, the review stage, and the discussion stage. In the first stage, each annotator annotated a set of 316 rename pairs of method names. The annotator would first split the old and new method names into their individual terms before annotating each term in the old and new names. Following the annotation process, we conducted a review stage. In this stage, the annotated datasets were exchanged between the annotators for review. If the reviewer did not agree with a specific annotation, the instance was marked for discussion. Finally, in the discussion stage, the annotator and reviewer discussed and looked at resolving conflicts. Instances where there was no consensus were discarded. During the entire process, the annotators had access to the source code and commit diff of the file containing the renamed method to refer. In total, we discarded 17 instances, leaving us with 615 annotated instances for our analysis.

RQ 1.1: What are the most common grammar patterns before and after a rename?

For this sub-RQ, we look at the frequently occurring grammar patterns for test methods in the annotated dataset (i.e., 615 method renames). These patterns are the complete/full grammar patterns for the names of test methods. Table 7.1 shows the top five patterns for old and new names, independent of one another. Below, we elaborate on common grammar patterns.

V NM+ N is also known as a verb phrase pattern; a phrase composed of a verb followed by a noun phrase. Typically, the verb represents the action to be applied to a head-noun that exists within the same phrase; typically the rightmost noun. In test methods, we observe that the term ‘test’ frequently represents the verb. Developers utilize the noun modifier (i.e., adjective) to specify characteristics or context around the entity being tested (i.e., the entity under test). An example of this pattern is the test method `testStringEncryption` [8]. The term ‘test’ represents the verb or action of the method. The term ‘Encryption’ is the head-noun or the entity under test, while the term ‘String’ represents the noun-modifier; descriptive of the entity under test.

The next two patterns: **V N** and **V V NM N** are both derivative verb phrases, where **V N** is a verb phrase with no adjectives and **V V NM N** is a verb phrase with an extra verb. Again, the first **V** is typically the word 'test' or a related term (e.g., can, should; we discuss this later). An example of the **V N** pattern is `testParser` [73], the action is ‘test’, while the term ‘Parser’ represents the object the action is applied to.

Table 7.1: Top five frequent grammar patterns for old and new names.

Grammar Pattern	**Count**	**Percentage**
Old name grammar pattern		
V NM N	41	6.67%
V N	27	4.39%
V NM NM N	25	4.07%
V V NM N	22	3.58%
V	20	3.25%
Others	480	78.05%
New name grammar pattern		
V NM N	44	7.15%
V N	29	4.72%
V NM NM N	29	4.72%
V	17	2.76%
NM N	14	2.28%
Others	482	78.37%

The last pattern is **V**: This pattern occurs more frequently in test methods than in production methods. In production code, these methods have generic names (e.g., 'sort') [177] since they tend to represent generic functionality. However, in test code, the methods falling in this category are part of a test fixture[1] (i.e., a setup or teardown method). For example, the `setup` method is utilized by developers to initialize the environment for the test methods in the test suite [56].

As part of our analysis, we also look for patterns between the terms in the method's name and statements in the method's body. These observations we encounter can be beneficial to static analyzer-based code quality tools. These include using the 'Assert.fail' method when the method name contains the term 'fail' or 'failure' (e.g., `failPrefixMissing` in [22]). Further, the use of the terms 'true' and 'false' in the method's name is very likely to be associated with using the methods 'assertTrue' and 'assertFalse' in the method's body, respectively (e.g. `testUntilTrueDefinitionOnReducedPath` in [82]).

Based on results of our prior study [177] and our study, verb phrases (e.g., **V NM+ N**) are the most common grammar pattern for method names regardless of whether they are test or production names. Thus, in this sub-RQ, we find no significant difference in the most *frequent* test

[1]A test fixture is utilized by developers to eliminate duplicate code and ensure a fixed environment for the tests.

Table 7.2: Occurrence of test naming patterns in test and production code.

Wu and Clause's Test Pattern Name	Grammar Pattern	# of Old & New Test Method Instances	% of Test Method Instances Preserved After Rename	# of Production Method Instances	Example
Is and Past Principle Phrase	V V+	353	62%	6	`testGetActions` [46] 'test' and 'Get' are verbs
Dual Verb Phrase	V V N+	52	46%	0	`testFindResourceByName` [72] 'test' and 'Find' are verbs, while 'Resource' is a noun
Verb Phrases With(out) Prepended Test	V N V+	29	67%	3	`testFormUploadLargerFile` [60] 'test' is a verb, while 'Form' is a noun and'Upload' is a verb
Divided Duel Verb Phrase	V N V N+	2	0	0	`testUidFetchBodyPeek` [81] 'test' and 'Fetch' are verbs, while 'Uid' and 'Body' are nouns
Noun Phrase	N	5	0	3	`main` [66] 'main' is a noun
Verb With Multiple Nouns Phrase	V N N N	–	–	0	Not observed in our dataset of annotated test methods

and production method grammar patterns. However, we also found that approximately 39.29% of test method grammar patterns are unique (i.e., they only occur once); in contrast to 24.72% of unique production method grammar patterns [177]. This difference implies that there may be a more diverse population of patterns in test methods. We address this in the next sub-RQ.

RQ 1.2: How do grammar patterns in test methods compare to defined naming patterns for test and production methods?

While our findings from the first sub-RQ show us that there are a small number of very frequent grammar patterns which are common to both test and production methods, it also indicates that there may be a difference in the diversity of these patterns. Because we want to understand what grammar patterns tell us about the similarity and differences in test and production methods, we use this sub-RQ to explore common grammar pattern prefixes for test methods; instead of only looking at the full grammar pattern as we did in the prior sub-RQ. By loosening the constraint to allow partial (i.e., prefix) grammar patterns, we aim to understand the diversity of grammar patterns in test methods.

To this end, we compare the catalog of test method name patterns formulated by Wu and Clause [222] against our annotated dataset, and also examine the occurrence of these patterns in production method names discussed in our prior study. [177]. While we compare to Wu and Clause's work, their goals were somewhat different. Their patterns are primarily *prescriptive*; creating templates that developers should use to improve test names. Our work is *descriptive*; attempting to examine the structures latent in test names while not prescribing what developers should use. Even so, the patterns Wu and Clause create are based on testing patterns they observed, and so it is appropriate to relate our patterns to theirs. The difference between our patterns and theirs can be seen in

Table 7.2. The leftmost column contains Wu and Clause's pattern names. To its right is another column showing the grammar pattern that corresponds with Wu and Clause's named patterns. Wu and Clause abstract away some detail (i.e., the tail of our grammar patterns) to necessarily and effectively discuss general naming patterns and their semantics. In this study, we keep these details, which causes several of our patterns to fit into a single one of Wu and Clause's patterns due to being derivative of a high-level pattern they already identified. However, this helps us understand how some of the granular differences that do not appear in Wu and Clause's work affect test name semantics.

In Table 7.2, we also show the frequency of Wu and Clause's patterns in our data, the number of production methods with the corresponding pattern, and the percentage of these patterns, which were conserved after a rename was applied. Where applicable, the '+' symbol, in Table 7.2, indicates that other POS tags precede and/or follow the pattern. One thing to highlight about this table is that we did not find the 'Verb With Multiple Phrases' pattern in our dataset, which corresponds to a grammar pattern of **V N N N**. Part of the reason for this is likely because we used a different tagset than Wu and Clause, who do not appear to use noun modifiers (NM). However, we did not want to assume their tagset and did not find a definition for the tagset they used in their study. Based on our understanding of their patterns, **V N N N** for them is the same as **V NM NM N** for our grammar patterns. Also, though we report the frequency at which our patterns match Wu and Clause's, it is important to remember that the tagsets in our studies may not completely match up. This does not matter for our study; we are not trying to determine the legitimacy or frequency of their test patterns. Instead, our work is aiming to find patterns that Wu and Clause may have overlooked and to add further legitimacy to their findings.

The grammar pattern prefixes we find are mostly derivatives of those found by Wu and Clause. However, there are several grammar patterns in our dataset that differ in interesting ways. We discuss these now.

V V N P+: This pattern is similar to Wu and Clause's 'Dual Verb Phrase' pattern. The primary difference is the presence of a preposition. For example, in the name `testReadFileFromClasspath`, 'test' and 'Read' are verbs, 'File' is a noun, and 'From' is a preposition [61]. Approximately 43% of the renames contained the prefix in the old and new names. Some of the common prepositions utilized by developers include 'of', 'with', and 'to'. Prepositions show a relationship between words, such as when and where things are related to each other. The preposition in this pattern is important because it identifies the relationship between the noun phrases on either side of it. We can use the preposition to assess the quality of a name based on which preposition is used, and whether the behavior of the test supports the use of the provided preposition. For the example above, using static analysis, we can check for the use of a file read operation that leverages the classpath. Normally, this might be difficult, but there is a finite number of prepositions in English (i.e., developers do not create new prepositions on the fly), meaning the behavior they describe is generally well-defined and finite.

N V+: This pattern consists of a noun followed by a verb (e.g., in `projectClosed`, 'project' is a noun, and 'Closed' is a verb [17]). Looking at the set of production methods, we observe ten instances of methods with this prefix pattern. From our annotated dataset of test methods, we observe 22 rename instances of this prefix. Additionally, approximately 64% of the renames contained the prefix in the old and new names.

+VM+: While not strictly a prefix grammar pattern, we include this observation in our findings since 1) verb modifiers have not been discussed at length in prior literature, 2) we found several patterns containing adverbs in our dataset, and 3) it is possible to use some of our observations about naming and implementation practices based on the presence or absence of certain adverbs. We start with an example: in the name `test_get_NotExisting`, 'Not' is an adverb [62]). We encounter 86 rename instances containing one or more adverbs in the name. Furthermore, we notice that developers utilize the same adverb from the old name in the new name when performing a rename of the method 78% of the time. Additionally, the top three terms associated with an adverb are 'not' (26 instances), 'when' (25 instances), and 'exactly' (5 instances). When combined with static code analysis, our observation becomes useful as it helps in appraising the name of an identifier. For instance, when examining the source code, we observe that method names containing the adverb 'not' are typically associated with some form of null based checking (e.g. use of 'assertNull' in the method `test_get_NotExisting` [62] and the use of 'assertNotNull' in the method `deleteindexNotExists` [27]). Finally, looking at production methods, we encountered seven instances of methods using this POS within its name.

+DT+ : Our rationale for the analysis of determiners is similar to our analysis of the **+VM+** pattern. Our dataset contains 72 instances that contain determiners in either the old or new name. From this set, there are 42 instances where the developer uses the same determiner in the old and new name (e.g., the term 'All' is preserved in the rename `findAllWithGivenIds` → `findAllWithIds` [45]). Regarding terms, the top three popular determiners are 'the', 'no', and 'all'. In terms of static code analysis, we observe that the term 'all' frequently co-occurs with collection-based data types in the method body (e.g., the use of 'List<Long>' in the method `testExecuteAll` [23]). The static analysis accuracy can be further improved by incorporating

Table 7.3: Top five frequently occurring pairs of complete grammar patterns for renamed unit test methods.

Rename Grammar Pattern		**Count**	**Percentage**
Old Pattern	**New Pattern**		
V NM N	V NM N	14	2.28%
V NM NM N	V NM NM N	9	1.46%
V	V	7	1.14%
V N	V N	7	1.14%
V NM N	V NM NM N	7	1.14%
Other Patterns		486	92.85%

findings from our prior study [192], specifically the findings on collection-based data types and singular/plural term changes.

Using grammar patterns, we have confirmed the existence of several naming patterns introduced by Wu and Clause. In addition, we identify patterns that were not identified in Wu and Clause's original set of patterns. The patterns we present are not frequent in production method names based on prior research, indicating that they are specific to test method names.

RQ 1.3: What are the most common grammar patterns before and after a rename?

In this sub-RQ, we examine the evolution of grammar patterns (i.e., the change in the grammar pattern when a method is renamed). In summary, our annotated dataset of 615 rename instances contained 168 (or approximately 27.32%) rename instances that did not show a change in grammar (i.e., the old and new grammar patterns were the same). Represented in Table 7.3 are the top five frequently occurring complete grammar pattern pairs. However, looking at the number of instances associated with each pair, we observe a low count (the most being 14 instances). Furthermore, our dataset contained 446 instances of grammar pattern pairs that occurred only once. This phenomenon (i.e., a wide variety of grammar patterns) highlights the diversity of our dataset and, therefore, impacts our analysis of rename pairs. Therefore, for the same purpose as RQ 1.2 we use prefix patterns to perform our analysis. We extracted frequently occurring pairs of rename prefixes for patterns where either the old or new name consists of prefixes of length two, three, four, or five. From this data, we show the top three frequently occurring pairs in Table 7.4.

From these tables, we make a couple of observations. The first is that renames do not typically change the POS tag of a word. Even when a word is changed, it is still the same type (i.e., at the POS level). Further, these renames follow the typical verb phrase method naming grammar pattern *V NM N* $\rightarrow$ *V NM N*. For example, in commit [8], when renaming the method `testStringEncryption` $\rightarrow$ `testStrongEncryption`, the POS is preserved even though terms in the name are changed; the terms 'String' and 'Strong' are considered as noun modifiers in this instance.

The second observation comes from Table 7.4. As the prefixes in this table increase, the original set of grammar prefixes remains the same. For instance, consider the two prefix pattern *V V* $\rightarrow$ *V V*, when the prefix pattern increases to three, the new pattern still retains the original prefix pattern: *V V NM* $\rightarrow$ *V V NM*. This observation remains consistent as prefixes increase to five prefixes. This shows that grammar pattern prefixes for test method names are consistent across renames. The primary takeaway from this sub-RQ is that grammar patterns are stable.

Summary. Using prefix grammar patterns of method renames, we obtained many interesting pattern changes to analyze. We performed this analysis in the context of prior work on test name templates. Our analysis confirms a number of the test name templates and also shows the existence of a few patterns that do not match up to any template provided in prior work. Particularly, patterns that include determiners, prepositions, and adverbs. We find that they have special, oftentimes implementation-oriented meaning in test method names. Finally, in RQ 1.3, we find that grammar pattern prefixes are stable; they do not change very often during rename activities.

7.3.2 RQ2: How are changes to the grammar pattern related to changes in the semantic meaning of the corresponding method name?

Approach: To determine the semantic change a name undergoes during a rename, we utilize a rename taxonomy defined by Arnaoudova et al. [101] and utilized in prior identifier rename studies [185,191,192] on our annotated dataset. This taxonomy helps us categorize renames into two categories– renaming form and semantic change. The renaming form looks at the terms added and removed to determine the complexity of the rename– simple, complex, reordering, and formatting. A rename is simple if only one term is added or removed. A complex change occurs if more than one term is added or removed. Reordering is when two or more terms switch positions. Finally, a formatting change occurs if the developer only makes a change in case or adds/removes a separator or number. In terms of semantic change categories, a rename can either preserve or modify the meaning of the name. A modification to a name can change, narrow, broaden, add or remove the meaning of the name.

From our set of 615 annotated instances, we observe that 291 (or approximately 47%) of the

Table 7.4: Top two frequently occurring pairs of two, three, four and five prefix grammar patterns for renamed unit test methods.

Rename Prefix Grammar Pattern		**Count**	**Percentage**
Old Pattern	**New Pattern**		
Two Prefix Pattern			
V V	V V	142	23.51%
V NM	V NM	103	17.05%
V N	V N	39	6.46%
	Other Patterns	320	52.98%
Three Prefix Pattern			
V V NM	V V NM	55	9.79%
V NM N	V NM N	36	6.41%
V NM NM	V NM NM	29	5.16%
	Other Patterns	442	78.65%
Four Prefix Pattern			
V V NM N	V V NM N	24	4.96%
V NM NM N	V NM NM N	19	3.93%
V V NM NM	V V NM NM	14	2.89%
	Other Patterns	427	88.22%
Five Prefix Pattern			
V V NM NM N	V V NM NM N	10	2.75%
V V NM N P	V V NM N P	9	2.48%
V V NM NM N	V NM NM N	4	1.10%
Other Patterns		340	93.66%

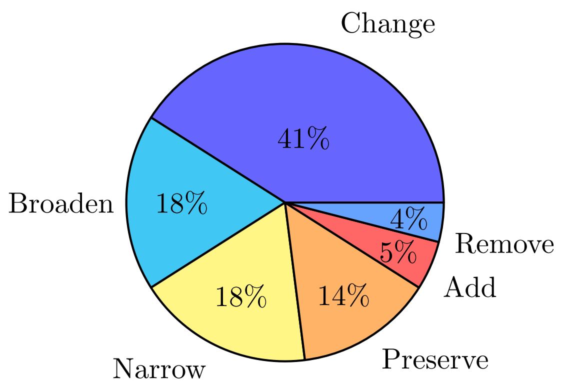


Figure 7.2: Proportion of semantic updates to 615 annotated test methods.

Table 7.5: Top five frequently occurring rename semantic updates for pairs of complete grammar patterns.

Rename Grammar Pattern		**Result**	**Count**	**Percentage**
Old Pattern	**New Pattern**			
V NM N	V NM N	Change	10	2%
V D D D	V D D D	Preserve	6	1%
V NM NM N	V NM NM N	Change	6	1%
V N	V NM N	Narrow	5	1%
V NM N	NM N	Broaden	5	1%
Other Patterns			583	95%

instances had a simple change, while 261 instances (or approximately 42%) had a complex change. From the semantic category, as depicted in Figure 7.2, we observe 255 (or approximately 41%) of instances had a change in meaning, while the narrowing and broadening categories each had approximately 18%. These findings are in contrast to prior research [185, 191], which shows that the majority of renames are of simple form and narrow in meaning. The prior studies utilize datasets comprising of mined rename refactoring operations of test and production source code files in Java projects. Looking at the set of renames categorized under preserve, we observe that the majority of these renames are due to developers either adding or removing numbers or underscore characters to/from the old name or performing a change of case (e.g., `test_13` → `test13` [65]).

Examining the change in meaning instances, 208 (or 33.82%) of the instances show an unrelated relationship between the old and new names. For example, in renaming `testLog` → `testEigenSingularValues` [77], there is no semantic relationship between the swapped terms. Approximately 7.15% instances

Table 7.6: Frequently occurring rename pairs of prefix patterns for different semantic categories.

Rename Grammar Pattern		**Semantic**	**Count**	**Percentage**
Old Pattern	**New Pattern**	**Type**		
		Two Prefix Pattern		
V V	V V	Change	70	28.46%
V NM	V NM	Change	50	20.33%
V N	V N	Change	15	6.10%
Other Change Patterns			111	45.12%
V V	V V	Preserve	20	22.99%
V NM	V NM	Preserve	14	16.09%
V N	V N	Preserve	13	14.94%
Other Preserve Patterns			40	45.98%
V V	V V	Add	7	30.43%
V NM	V NM	Add	4	17.39%
N V	N V	Add	2	8.70%
Other Add Patterns			10	43.48%
V V	V NM	Remove	7	23.33%
V V	V V	Remove	4	13.33%
V N	V N	Remove	2	6.67%
Other Remove Patterns			17	56.67%
V V	V NM	Broaden	19	17.27%
V NM	NM N	Broaden	9	8.18%
V NM	V NM	Broaden	9	8.18%
Other Broaden Patterns			73	66.36%
V V	V V	Narrow	32	29.63%
V NM	V NM	Narrow	24	22.22%
V N	V NM	Narrow	8	7.41%
Other Narrow Patterns			44	40.74%
		Three Prefix Pattern		
V V NM	V V NM	Change	30	13.16%
V NM N	V NM N	Change	21	9.21%
V NM NM	V NM NM	Change	17	7.46%
Other Change Patterns			160	70.18%
V V NM	V V NM	Preserve	11	14.67%
V NM N	V NM N	Preserve	7	9.33%
V D D	V D D	Preserve	6	8.00%
Other Preserve Patterns			51	68.00%
V NM N	V NM N	Add	2	8.70%
V NM N	V NM NM	Add	2	8.70%
N NM	N NM P	Add	1	4.35%
Other Add Patterns			18	78.26%
V V NM	V NM NPL	Remove	3	10.00%
V V NM	V NM N	Remove	2	6.67%
DT NM NM	NM NM N	Remove	1	3.33%
Other Remove Patterns			24	80.00%
V NM NM	V NM N	Broaden	8	7.69%
V V NM	V NM N	Broaden	7	6.73%
V V NM	V NM NM	Broaden	7	6.73%
Other Broaden Patterns			82	78.85%
V V NM	V V NM	Narrow	11	10.78%
V NM N	V NM NM	Narrow	6	5.88%
V N V	NM N	Narrow	5	4.90%
Other Narrow Patterns			80	78.43%

contained more than one type of relationship between the old and new names. For example, in renaming `testDeserializeExpandCharge` → `testDeserializeWithExpansions` [58], we observe an addition and removal of terms as well as a change in plurality. Finally, three (or 0.49%) instances exhibited an antonym relationship as in case of renaming the method `shouldAcceptRaxProtocols` → `shouldRejectRaxProtocols` [13]; here we see the term 'Reject' replacing 'Accept'.

In Table 7.5, we provide the top five rename forms and semantic updates associated with a complete grammar pattern pair. From this table, we observe that the most frequent grammar pair, *V NM N* → *V NM N* is mostly associated with a change in meaning. For example, in the rename commit [8], `testStringEncryption` → `testStrongEncryption`, a single term is replaced making it a Simple form type change and since there is no semantic relationship between the terms 'String' and 'Strong' it is categorized as a general change in meaning. However, from this table, we observe a low volume of instances of grammar patterns associated with the semantic categories; this behavior is similar to what is observed in RQ 1.3. Hence, similar to RQ 1.3, going forward, we look at the relationship between prefix grammar patterns and name semantics. Presented in Table 7.6, we provide the top three frequently occurring prefix patterns for each semantic category.

From Table 7.6, we observe that the rename prefix pattern *V V* → *V V* is associated with all semantic categories. However, it is more prevalent with the change in meaning category. This same prefix pattern is also the most common rename pattern, as reported in RQ 1.3. From the table, we observe that remove and broaden meaning shows a divergence in the prefix pattern; the most frequently occurring prefix pattern for these two categories is *V V* → *V NM.* For the broadening pattern, we observed that in the majority of developers tend to remove the term 'test' from the old name (e.g., `testPinnedExternals` → `pinnedExternals` [59]).

As the number of prefixes increases, the volume of these instances being associated with a semantic category decreases. Again, similar to RQ 1.3, this phenomenon will help determine the quality of a test method's name either when a developer performs a rename or during static analysis of code. However, as these are prefix patterns, it should be noted that terms associated with the POS tags in the prefix might not always be the terms contributing to the semantic transformation of the method name. Hence, the findings from this RQ should be used in conjunction with findings from other RQs and also prior work, such as [192] study of identifier renaming using data types and co-occurring refactorings.

Summary. Our analysis of test methods shows that developers frequently change the meaning of a test method's name when performing a rename. This contrasts with prior research, which studied production and test names together, finding that these tend to narrow in meaning. One conclusion we may draw from this is that test methods more frequently change in meaning than the general population of methods. Another potential explanation is that it is more challenging to analyze the relationship between words in test methods. If word relationships in test methods are heavily domain-driven, then some of the underlying technology, such as WordNet, used to analyze these may not work well. More research is needed to conclude which case is valid. However, whichever case we are in, it is clear that the relationship between words in test methods as they evolve is different from the general population of methods. Thus, recommending test name structures or words will potentially require specialized approaches trained specifically on test naming structures.

7.3.3 RQ3: What are the most common term changes, and what is the relationship between the added term and removed term?

Approach: In this RQ, we examine the frequent terms added to and removed from test methods due to a rename. The experiment in this RQ utilizes the complete dataset of test method names. We first utilize the heuristic splitter algorithm implemented in the Spiral package [146] to determine the terms that form a name. Next, for each rename instance, we extract only the terms that were added and removed. Finally, for each added and removed pair of terms, we count the number of times the pair exists in the dataset. For example, when *getEmployeeName* is renamed to *testEmployeeLastName*, the added terms are ‘test’ and ‘Last’, while the removed term is ‘get’. We search our dataset for the occurrence of ‘test’ & ‘get’ and ‘Last’ & ‘get’. Additionally, as part of our qualitative approach, we manually analyzed a statistically significant sample that comprises of the top 646 frequently occurring pairs. The sample represents a 99% confidence level and a 5% confidence interval from our population of 21,615 pairs of added and removed terms. As part of this analysis, we annotated the semantic relationships between the added and removed terms. The semantic annotations include: synonyms, antonyms, specializations, and generalizations.

Analyzing the list of 646 removed-added pairs, we observe instances where the developer either adds or removes numerical digits to or from the replacement term. An in-depth look at these identifiers shows that a vast majority of such names usually do not contain any other terms that describe the behavior of the test method (e.g., `test15_6_5` → `test16_9_5` [71]). Most likely, these are auto-generated tests or tests utilized for debugging purposes. To facilitate the use of English semantic rules to determine the relationship between the term pairs, our analysis of term pairs will be limited to only pairs that do not have numerical digits. Looking at the top five frequently occurring term pairs, we observe developers replace ‘has’ with ‘contains’ (94 instances), ‘test’ with ‘can’ (58 instances), ‘all of’ with ‘at least’ (46 instances), ‘with’ with ‘when’ (40 instances), and ‘test’ with ‘should’ (38 instances). Additionally, we also observe that developers frequently replace the term ‘test’ with a term associated with a Boolean return type (e.g., ‘can’, ‘is’, ‘should’).

Next, we look at the different types of semantic relationships between the removed-added pairs. From our dataset, we observe that 294 of the removed terms were replaced with terms of the same POS. For example, in renaming `testFilterBaseNice` $\rightarrow$ `testSelectBaseNice` [69] the developer replaces the term ‘Select’ with ‘Filter’, both of which are verbs. The majority of replacement terms were added in the same position as the removed term in the name. Looking at the types of semantic relationships in the dataset, we observe 36 pairs of terms as synonyms (e.g., `boundingCube` $\rightarrow$ `boundingBox` [80]) and 22 pairs having an antonym relationship (e.g., `genericExtension` $\rightarrow$ `specificExtension` [78]). We also identified 12 instances each of specialization (e.g., `testPredictions` $\rightarrow$ `validatePredictions` [79]) and generalization (e.g., `listContains` $\rightarrow$ `collectionContains` [68]).

Looking at the root (i.e., stem) of the removed-added term pairs, we observe that 70 pair instances have the same stem. For example, in the following rename `testTwippleUploader` $\rightarrow$ `testTwippleUpload` [79], the terms ‘Uploader’ and ‘Upload’ have the same stem– ‘upload’. We also observe 17 instances of tense change (e.g., `isOrderedFailure` $\rightarrow$ `isInOrderFailure` [57]) and nine instances of plurality changes (e.g., `enqueueJob` $\rightarrow$ `enqueueJobs` [64]), and spelling corrections (e.g., `projectVisitorIsInkvoked` $\rightarrow$ `projectVisitorIsInvoked` [32]) each.

Our manual analysis shows that determining the relationship between terms is a challenging task due to the diverse ways in which developers rename identifiers. We made the following observations during our qualitative analysis: (1) although proper naming helps understand what the test verifies and how the underlying system behaves, some terms are ambiguous, which makes it challenging to determine the semantic relationship between the pair due to the use of domain terminology (e.g., ‘LBDevice’ is replaced with ‘Zeus’ [12]), (2) multiple terms can replace a single term and vice versa; this type of change is done due to specialization/generalization of behavior or in situations where the names are synonyms (e.g., the terms ‘not started’ are replaced by ‘closed’ [67]), and (3) the terms are unrelated (e.g., ‘Latency’ is replaced with ‘Metrics’ [74]).

Finally, when examining the code, we observe that specific terms in a method’s name can indicate the presence of specific statements in the body of the method. For instance, we observe that the presence of the terms ‘at least’, ‘all of’, or ‘all’ acts as a sign that a method performs tests on collection-based objects such as List, Map, or custom collection types. For example, the method `findAllWithGivenIds` contains a collection object that is subject to a series of tests (i.e., assertion statements). Similarly, the occurrence of the term ‘exception’ indicates that the purpose of such methods is to verify that an exception occurs as part of the execution of the test. In such instances, developers either utilize the 'expected' parameter as part of the `Test` annotation or places an assertion statement in the catch section of the try-catch block that handles the exception that the developer expects to be thrown (e.g., `invokingStaticMethodQuietlyShouldWrapIllegalArgume ntException` [63]). These observations show how static analysis combined with NLP techniques can support the automation of identifier name appraisal algorithms.

Summary. When replacing terms in a method's name, developers frequently preserve the overall meaning of the method name by utilizing a synonym of the removed term. In addition, there are some interesting common word and phrase substitutions we observed in this set. Including 'has' $\longleftrightarrow$ 'contains' and 'all of' $\longleftrightarrow$ 'at least'. Many of these can be linked with code semantics. For example, 'all of' changing to 'at least' indicates a shift in testing behavior; instead of testing for the presence of all entities, they are testing for a subset. We manually confirmed that some of this behavior can be directly mapped to code changes, and thus we may be able to provide some naming recommendations in the future based on these trends. In addition, the term 'test' is frequently swapped with terms such as 'can', 'is', and 'should'. The relationship between these terms and the term 'test' range from synonyms to metonyms.

Chapter 8

Understanding Digits in Identifier Names: An Exploratory Study

The contents of this chapter are part of the study "***Understanding Digits in Identifier Names: An Exploratory Study***" published in the Proceedings of the 1st International Workshop on Natural Language-based Software Engineering [194].

8.1 Introduction

As lexical tokens that uniquely identify entities in the code (such as classes, methods, variables, etc.), identifier names play an essential part in program comprehension activities, with well-constructed names improving comprehension activities by an estimated 19% [144]. Ideally, for identifier names to assist developers with understanding the code, the name must be unambiguous, and intent-revealing in communicating the purpose and behavior of its associated source code [116]. However, with developers having the ability to craft names using a variety of terms [131,177,190], it is challenging to ensure consistent quality of the identifiers in the source code. This is exacerbated by the fact that around 70% [124] of the characters in the code base are dedicated to identifier names. Ignorance of the quality of identifier names is ignorance of the quality of a large portion of the code.

To correct low quality identifiers, developers rename [133] them. Rename refactorings are defined as refactorings that modify the name of an identifier without modifying the intended behavior of the code. Many Integrated Developer Environments (IDEs) offer a built-in rename refactoring functionality. Most of these IDEs only support the mechanical act of renaming; they allow a developer to choose what identifier they want to rename, accept the new name should be used, and then perform checks to avoid name collisions. There is little or no support beyond this to assist developers in determining when and how (e.g., help them pick terms). This is in spite of the fact that renaming is one of the most common refactorings [184,191]; further highlighting that naming is a severe problem in software maintenance.

To understand the characteristics associated with high-quality names, past work in this area has focused on both empirical and developer-centric studies. To this extent, research studies have surveyed professional developers [95], examined the presence, significance, and influence of abbreviations [148,149,174,176], acronyms [117], the comprehensibility [105,144,206] of different types of names, and investigated the semantic evolution of the identifiers [101,192]. However, these existing studies focused on the words that make up identifiers, not digits. Therefore, this study expands our understanding of identifier names by studying the digits that appear within them. Our work aims to help lay the foundation in this area of research by conducting an exploratory study on the presence and purpose of digits in an identifier's name.

The goal of this study is to understand the structure of identifier names containing digits and the semantics expressed by digits in identifier names. To this extent, we study *the part played by digits in identifier names* by examining the name's evolution and the meaning conveyed by the digit to the overall purpose of the identifier. We envision findings from our study supporting the development of tools and techniques in identifier name recommendation and appraisal and the auto-generation of comprehension-friendly source code. We answer the following research questions (RQs):

RQ1: How does identifier renaming operations in the source code impact the existence of digits in an identifier's name? This question explores the extent to which digits are present in identifier names and how they change over time. Findings from this RQ inform us of the volume and characteristics of such identifiers; including if digits are generally preserved after a rename has been applied, and the number of digits that typically occur in a name.

RQ2: How do developers utilize digits in an identifier's name to convey meaning? In this RQ, we perform a qualitative examination of the terms in an identifier's name and its related code to determine the rationale behind the presence of digits in an identifier's name. Through our analysis, we establish a taxonomy for the presence of digits in an identifier's name.

8.2 Methodology

This section provides details about the methodology of our study. Depicted in Figure 8.1 is an outline of our experiment design. We perform a series of natural language-based processing activities on the source dataset before answering our RQs. In the below subsections, we describe these

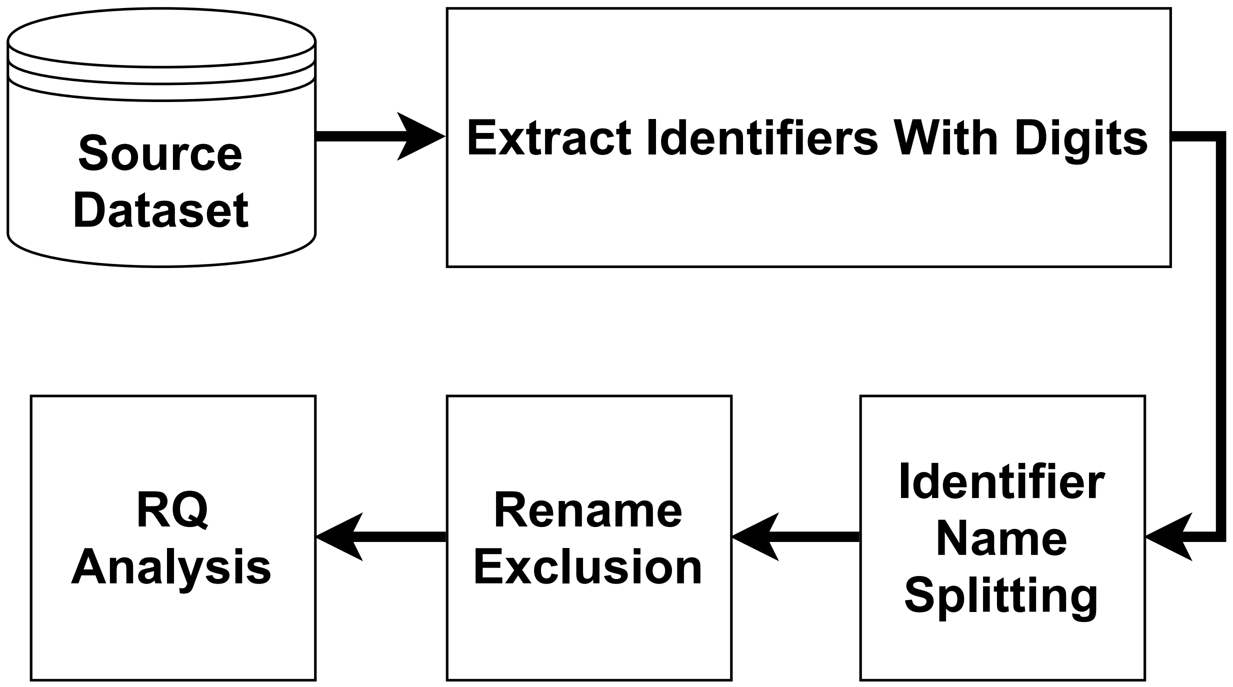


Figure 8.1: Overview of our experiment design.

activities.

8.2.1 Source Dataset

In this study, we utilize the dataset of rename refactorings made available from a prior study of ours [190]. The dataset contains the commit history and refactoring operations of 800 open-source Java systems. The refactoring operations were mined at the commit level using RefactoringMiner [218], a state-of-the-art tool with a precision of 98% and a recall of 87% [208, 219], and contain the rename operations performed on classes, attributes, methods, parameters, and local-variables. As shown in Table 8.1, in total, the dataset contains 428,079 rename operations and 926,948 non-rename operations. The dataset also indicates if a source code file is a JUnit-based unit test file or not by following an approach similar to [187].

8.2.2 Extract Identifiers With Digits

As our study is limited to analyzing only renames that involve digits in the identifier's name, our initial step is to extract such rename operations from the source dataset. We achieve this by using

Table 8.1: Volume of rename and non-rename refactoring operations in the source dataset.

Refactoring Operation	**Count**	**Percentage**
Rename Attribute	140,306	10.35%
Rename Method	91,235	6.73%
Rename Variable	89,032	6.57%
Rename Parameter	75,042	5.54%
Rename Class	24,556	1.81%
Move And Rename Class	7,676	0.57%
Move And Rename Attribute	232	0.02%
Non-Rename Operations	926,948	68.41%
Total	1,355,027	100%

a regular expression to detect the presence of a digit in either the old or new name of the rename operation; the results of this activity yield 149,980 rename operations.

8.2.3 Identifier Name Splitting

A prerequisite to analyzing identifier names is to split the name into its constituent terms. To perform this activity, we utilize the Ronin splitter algorithm implemented in Spiral, a specialized open-source identifier name splitting Python package [146]. The splitter utilizes heuristic rules, English terms, and token frequencies to determine the individual terms in an identifier's name. Furthermore, prior studies have utilized this package to split identifier names [190, 191, 192]. As an example, the name 'service2WsdlResource' is split into'service', '2', 'Wsdl', and 'Resource'.

8.2.4 Rename Exclusion

Domain Terms:

As part of the identifier name splitting activity, the splitter considers standard domain terms that contain digits and refrains from splitting such terms. For example, the splitter splits the name 'testi18nGetAll' into the terms 'test', 'i18n', 'Get', and 'All' since 'i18n' is an established computing abbreviation associated with internationalization and localization. Hence, we exclude such rename operations from our analysis.

Table 8.2: Renames distribution by identifier type.

Identifier Type	**Count**	**Percentage**
Variable	4,319	28.00%
Method	3,545	22.98%
Parameter	2,878	18.66%
Class	2,779	18.02%
Attribute	1,903	12.34%
Total	15,424	100%

Auto-Generated Code:

A manual examination of the rename instances in our dataset showed the excessive presence of automatically generated code. For example, we encountered identifiers like 'LA642_0' and 'FOLLOW_EOF_in_entryRuleCodeRef1107'. Such code occurs in projects that utilize parser generator frameworks like ANTLR. Identifier names generated automatically are a significant portion of identifiers with numbers in them. Because we wanted to get a good sample of identifiers that included digits but were written by humans, we used heuristics to remove many auto-generated identifiers from our set. To determine how to exclude such source code, we analyzed a statistically significant sample of 383 rename instances (confidence level of 95% and an interval of 5%) and determined some common traits we could use to remove them. This helped us get a better, more equal sample that included all identifiers, including some auto-generated identifiers that we were unable to easily exclude.

8.2.5 RQ Analysis

To answer our RQs, we analyze 15,424 rename operations. This value comprises of 2,683 renames in unit test files and 12,741 renames in non-test files. Furthermore, as shown in Table 8.2, method variables accounted for most digit-based renames with 28% instances, followed by method names at 22.98%. We follow a mixed-methods [215] approach to answer our RQs. In this approach, we utilize well-established statistical measures and custom code to report trends and patterns in our dataset and manually analyze the data to present representative examples to complement our quantitative findings. We created these research questions with the purpose of understanding both a quantitative and qualitative perspective on the appearance, and purpose, of digits when they appear in identifiers.

Table 8.3: Top five distribution of the number of digits in the old and new names.

Number of digits in the old name	**Number of digits in the new name**	**Count**	**Percentage**
1	1	5,370	79.93%
2	2	518	7.71%
3	3	207	3.08%
1	2	185	2.75%
2	1	92	1.37%
Other combinations		346	5.15%
Total		6,718	100%

Table 8.4: Top five distribution of the position of digits in the old and new names.

Position of digits in old name	**Position of digits in new name**	**Count**	**Percentage**
2nd	2nd	1,930	28.73%
3rd	3rd	757	11.27%
4th	4th	432	6.43%
5th	5th	328	4.88%
2nd	3rd	283	4.21%
Other combinations		2,988	44.48%
Total		6,718	100%

8.3 Experimental Results

In this section, we report on the findings of our experiments by answering our RQs. The first RQ is primarily quantitative and examines the structural evolution of the name with respect to the digits in the name. The second RQ examines how digits convey meaning in an identifier's name. The tables in the RQs show only the most frequently occurring instances.

Table 8.5: The most frequently occurring instance of digit position and count in test and non-test files.

Number of digits in the old name	**Number of digits in the new name**	**Count**	**Percentage**
	Test Files		
1	1	1,126	82.73%
	Non-Test Files		
1	1	4,244	79.22%
Position of digits in old name	**Position of digits in new name**	**Count**	**Percentage**
	Test Files		
2nd	2nd	283	20.79%
	Non-Test Files		
2nd	2nd	1647	30.74%

Table 8.6: Top five distribution of the position of digits and the number of digits in the old and new names.

Position of digits in old name	**Position of digits in new name**	**Number of digits in the old name**	**Number of digits in the new name**	**Count**	**Percentage**
2nd	2nd	1	1	1,930	28.73%
3rd	3rd	1	1	757	11.27%
4th	4th	1	1	432	6.43%
5th	5th	1	1	328	4.88%
2nd	3rd	1	1	283	4.21%
Other Combinations				2,988	44.48%
Total				6,718	100%

8.3.1 RQ1: How does identifier renaming operations in the source code impact the existence of digits in an identifier's name?

In this RQ, we take a quantitative approach to study the treatment of names with digits over time. To this extent, we examine the presence and/or absence of digits in an identifier's name before and after a rename operation. Findings from this RQ will give insight into the prevalence of digits, the digits frequently utilized in an identifier's name, and the action taken on these identifiers.

From the set of 15,424 renames with digits, we observe that 6,718 (or 43.56%) instances have a digit present in the old and new name (e.g., 'h1' → 'gain1'). Renames where all digits are removed from the name account for 5,135 (or 33.29%) instances (e.g., 'arg1' → 'id'), while 3, 571 (or 23.15%) instances show the adding of digits to a name that did not contain digits (e.g., 'log' → 'log1').

Our analysis now focuses on the instances where digits are preserved in a rename. First, when comparing the number of digits in the old and new name, we observe that 6,170 or 91.35% instances have an equal number of digits in the old and new name. Additionally, from this list, we observe that most instances (5,370, or 79.93%) contain only one digit in the old and new names. The next highest set of names contains two digits in the old and new name with 518 or 7.71% instances (e.g., 'april7th2011' → 'april8th2011'). Table 8.3 shows the distribution of the top five combinations of the number of digits in old and new names. Our subsequent analysis looks at the position of the digits occurring in the names. Findings show that most renames (i.e., 4,402 or 65.53% instances) preserve the position of the digits in the new name. Furthermore, a digit will likely appear in the second position of the old and new names (e.g., 'node2' → 'node3').

Table 8.4 shows the distribution of the top five combinations of the positioning of digits in the old and new names. Furthermore, when comparing renames in test and non-test files, the most occurring rename combination regarding the number of digits and digit position is the same for both file types. Table 8.5, shows the frequently occurring instance for each file type. Finally, a combination of digit count and position (refer Table 8.6) shows that identifier names typically use a single digit that appears as the second term in the old and new name; we observe 1,930 or 28.73% instances of this pattern combination. We also observe that the single-digit occurrence in the names frequently happens regardless of the position of the digit.

Our final analysis, in this RQ, looks at the value of the digit appearing in the name. Looking at renames with digits in both names, our observations show that in most rename instances, the value '2' is present in the old and new names and is the only digit present in both names (e.g., 'slave2Index' → 'channel2Index'). This specific occurrence accounts for 1,052 or 15.56% instances.

The next highest occurrence is the value '1' in 851 rename instances as the only digit in the old and new name. Next, examining rename instances where at least one name contains a digit, we observe that the value '1' is either removed from the old name (e.g., 'arg1' → 'id') or added to the new name (e.g., 'oldDelta' → 'delta1') frequently with 1,563 and 1,512 instances, respectively. Finally, our observations show a diverse set of numeric values utilized in naming identifiers, resulting in a long tail when examining the frequency occurrence.

Summary for RQ1: When renaming identifiers containing digits, a frequent activity is preserving digits in the new name and usually using the same number of digits in the new name and preserving the position of the digits in the name. Furthermore, a single digit in the old and new name is the most commonly occurring pattern, with the digit usually occurring as the second term in the old and new name; a phenomenon common to test and non-test identifier names. Additionally, identifiers utilize a diverse set of digits in their name, with the number '2' being a common digit in the old and new name.

8.3.2 RQ2: How do developers utilize digits in an identifier's name to convey meaning?

The prior RQ was primarily quantitative-based, showing how digits evolve in an identifier's name. However, while the findings can contribute to the appraisal of identifier names, we do not know why developers utilize digits in identifiers and how these digits contribute to the overall meaning of the identifier. Hence, in this RQ, we manually examine the semantics of identifiers that include digits, and their surrounding source code, to determine the rationale for the presence of digits. This RQ aims to produce a taxonomy showing how digits convey meaning in an identifier's name. Our analysis is constrained to 375 rename instances, where either the old, new, or both names contain a digit. Furthermore, this dataset represents a stratified statistically significant sample; we utilized a confidence level of 95% and an interval of 5% for each identifier type (i.e., class, attribute, method, local-variable, and parameter). In the annotation process, we annotated the dataset with the rationale for the presence of the digit in the name. This was done by reading the identifier name and looking at the code associated with the identifier. Once this was complete, we compared their annotations and resolved any conflicts through discussion. Additionally, in specific instances where we encounter interesting phenomena, we perform a snowballing activity to locate examples of additional instances in the original dataset. We discuss taxonomy below and provide specific definitions in Table 8.7.

Table 8.7: Taxonomy showing how digits convey meaning in an identifiers name.

Category	Definition	Example
Auto-Generated	Identifier names in this category are generated by a code generation tool, framework, or IDE. The number in these identifiers may have significant meaning to the technique that was used to generate them. However, it is difficult to understand this meaning (if it exists) without a thorough understanding of the technique used to generate them. These identifiers are also typically not written in a way to support comprehension, since they are not typically maintained by developers directly but, instead, regenerated by the technique that created them in the first place.	*LA18_6*
Distinguisher	Identifier names in this category differ only by a digit, which is typically appended as the rightmost token. There should be at least two identifiers having a lexically identical name. The purpose of the digit is to avoid name collision at compile/parse time. The digit has no other significant meaning relating to the purpose of the identifier. Thus, the digit primarily operates to distinguish the identifier it is part of from other identifiers that are lexically identical other than their own digit.	*auditLog3*
Synonym	Identifier names in this category contain at least one digit utilized in place of a word. Sometimes digits are used to represent the meaning typically associated with certain words. The numbers 2 and 4 are very common examples of this, with 2 being associated with words like 'to' and 'too' while 4 is associated with words like 'for'. In this way, they function as a type of shorthand.	*convert2RList*
Version Number	Identifier names in this category contain at least one digit used to signify a version number. This typically means that the identifier represents an entity whose version is significant to its capabilities and limitations Version numbers were often used in the dataset to inform developers of what version of a framework, tool, protocol, etc, an identifier represented. This could, for example, be an identifier that represents an HTTP 1.0 request having the number 1_0 appended to it.	*V1DozerTransformModel*
Specification	Identifier names in this category contain at least one digit that represents a known specification. In many cases, this is a number that acts as a way to uniquely identify concepts, behaviors, or characteristics. These can be mathematical concepts, such as using 3D in identifiers that deal with 3-dimensional data; documented, project-specific behavior like in filter1_2, where the 1_2 in the identifier tells us which specific filter (i.e., 1_2) this identifier represents; and data characteristics such as 9 in arialRegular9Dark which gives us the size of the font data associated with the identifier.	*arialRegular9Dark*
Domain/Technology	Identifier names in this category have a digit that is part of the name of a domain term or technology. The digits themselves have no individual meaning besides the meaning endowed by the technology/domain that they are a reference to.	*resultDoubleExp4j*

Auto-Generated:

Identifiers falling under this classification are of two types - 1) part of a source file that is entirely auto-generated by a tool/framework (e.g., ANTLR) and 2) specific identifiers generated by the IDE in a source file containing developer-defined identifiers. The identifier names in tool/framework generated source files sometimes have the name of the data type or expression statement. However, like the integer variable 'LA18_6', this is not always the case. In the case of the latter, these identifiers are typically user interface (UI) controls. When a developer utilizes the IDE to drag-and-drop a UI control, such as a textbox, the IDE generates the code associated with the UI control (i.e., properties and event handler). The name of such identifiers starts with the type of control and ends with an incrementing digit. For example, 'jComboBox2' is the IDE generated name for a JComboBox UI control; the value '2' indicates that this is the second JComboBox control in the class. We also observe instances where developers rename auto-generated identifier names to more meaningful names, like 'jScrollPane4'→'scrollCompilerDescription'. The fact that some of these auto-generated names eventually received a higher-quality name indicates that auto-generated identifiers should not necessarily be discarded when performing this kind of research. In some cases, despite being auto-generated, they provide additional insight. The presence of these identifiers represents an opportunity for future research to study how developers use and modify auto-generated identifiers and how they differ, in terms of how they evolve, from other types of identifiers that originated from humans instead of code-generation technologies.

Distinguisher:

These types of identifier names are prevalent in our dataset. Developers create multiple identifiers in the source file with the same name, but each has a unique digit to avoid name collision at compile-time. Hence, the purpose of digits in these names is to distinguish them, lexically, from one another– hence their name. These types of names were noted but not studied in other research [174]. In most cases, the digit is the last term in the name, and developers increment the value of the digit with each new lexically-identical iteration of the identifier. An example of such an instance is the declaration of the variables 'auditLog1', 'auditLog2', and 'auditLog3', within the same method, where the digit is a distinguisher. A variant of this category is the use of generic names (and sometimes abbreviations) for identifiers. For example, we encounter method parameters named 'arg0' and 'arg1'.

Synonym

Developers utilize digits as synonyms for prepositions, with the value '2' frequently used as a substitute for the 'to' preposition. One such use of this digit is to indicate the result of an action or a process, such as naming transformation-based identifiers. Such identifiers either convert data from one format to another or hold the transformation results. For example, the method 'convert2RList' converts a table-based object, passed as a parameter, to a list-based object which it returns. Another use of the digit '2' is to indicate purpose or intention. For example, an attribute with the name 'mb2use' is utilized to hold the cache size that the program intends to use. We also encounter the use of the value '4' as a synonym for the preposition 'for' as in the method 'populate4Test', which instantiates a variable for use in test cases. Furthermore, such digits are contained within the name and not at the end of the name.

Version Number

Developers utilize identifiers to signify the version number the code supports. For example, looking at the comments in the class 'V1DozerTransformModel', the term 'V1' represents "A version 1 DozerTransformModel". A drawback of such an approach is the continual update of the identifier's name with new versions of the system, as shown by the renaming of the variable 'pg75' $\rightarrow$ 'pg80'.

However, certain developers also acknowledge this overhead and remove such dependencies as in the renaming of the identifier 'DROID_SIGNATURE_FILE_V45' → 'DROID_SIGNATURE_FILE' with the commit message "Changed variable name so that it is more generic than previously"

Specification

The digits in a name can relate to specifications of the system, such as size or dimensions, or provide details about the system's behavior. For instance, the value '9' in the attribute name 'arialRegular9Dark' indicates the size of the font object associated with the attribute. In another example, the comments for the class 'Geographic3DTo2DConversion' show that this class "Converts between a Geographic 3D and a 2D system". Though rare, we also encounter the use of digits to specify a specific use case, as in the name 'testUC_3_EraseFacetClassifier_NoSource'; this is a unique traceability mechanism, especially for test cases

Domain/Technology

The presence of digits is also due to developers using domain or technology names that contain digits. For instance, we observe developers utilizing API/library names as a term in the identifier's name, like 'Twitter4J', 'Neo4J', 'Slf4j', 'Log4j' , 'Args4j', Exp4j, and 'Junit4', such as in naming the attribute'resultDoubleExp4j'. The use of standards/formats in the name also results in digits in the name. For example, in the method name 'testCP437FileRoundtrip', the term 'CP437' is a standard for character encoding. Likewise, the value '1516' in 'parseInvalidHla1516eFomWithUndefinedTransportForAttribute' corresponds to an IEEE standard (i.e., IEEE-1516e).

Summary for RQ2: There are multiple reasons why an identifier name contains digits. Digits are frequently utilized to create unique identifier names; this is true for both auto-generated and developer-written source code. Developers also name identifiers based on domain and technology terms such as the API/library associated with the identifier, resulting in digits if the API/library name contains such values. Additionally, digits are also utilized to convey size, standards, versions, and words like 'to' and 'for'.

Research Focus Area: Tool Development

Chapter 9

IDEAL: An Open-Source Identifier Name Appraisal Tool

The contents of this chapter are part of the study "***IDEAL: An Open-Source Identifier Name Appraisal Tool***" published in the Proceedings of the 2021 International Conference on Software Maintenance and Evolution [189].

9.1 Introduction

Program comprehension is a precursor to all software maintenance task [200]; it is essential that a developer understands the code they will be modifying. Therefore, maintaining the internal quality of the code over its lifetime is of paramount importance. As fundamental elements in the source code, identifier names account, on average, for almost 70% of the characters in a software system's codebase [124] and play a significant part in code comprehension [121, 164]. Low quality identifiers can hinder developers' ability to understand the code [155, 206]; well-constructed names can improve comprehension activities by an estimated 19% [144].

However, there is still very little support for developers in terms of helping them craft high-quality identifier names. Research has examined the terms or structure of names [85, 124, 144, 191, 207] and produced readability metrics and models [112, 130, 205] to try and address this problem. However, they still fall short of providing tangible advice for improving naming practices in developers' day-to-day activities. The work we present in this study is designed to operate within an IDE, or a CLI, setting and provide real-time advice to developers about their naming practices.

Our work aims to provide the research and developer community with an open-source tool, IDEAL, that detects and reports violations in identifier names for multiple programming languages using static analysis techniques. In addition to identifying the identifier(s) exhibiting naming issues in the source code, IDEAL also provides necessary information for each reported violation so that appropriate action(s) can be taken to correct the issue. We envision IDEAL utilized by developers in crafting and maintaining high-quality identifier names in their projects and also by the research community to study the distribution and effect that various poor naming practices have in the field.

IDEAL is a multi-language platform for identifier name analysis. It is context-aware; treating test and production names differently since they have different characteristics [177, 190]. It allows for project-specific configurations and is based on srcML [120], allowing it to support multiple programming languages (specifically, Java and C#). IDEAL is publicly available as an open-source tool to facilitate extension and use within the researcher and developer communities.

Table 9.1 summarizes the linguistic anti-patterns currently detected by IDEAL. Anti-Patterns A.* to F.* are the set of original anti-patterns defined by Arnaoudova et al. [103], while the anti-patterns G.* are anti-patterns unique to IDEAL. We should also note that as an open-source tool IDEAL provides the necessary infrastructure for the inclusion of additional anti-patterns.

9.2 IDEAL Architecture

Implemented as a command-line/console-based tool in Python, IDEAL integrates with some well-known open-source libraries and tools in analyzing source code to detect identifier name violations. Depicted in Figure 9.1 is a view of the conceptual architecture of IDEAL. Broadly, IDEAL is composed of three layers– Platform, Modules, and Interface. It utilizes well-known tools and libraries used for natural language and static analysis, including Spiral [146], NLTK [108], Wordnet [168], Stanford POS tagging [216], and srcML [120].

9.3 Evaluation

To understand the effectiveness of IDEAL in correctly detecting identifier naming violations, we subjected IDEAL to two types of evaluation activities. First, we analyzed four popular open-source systems using IDEAL and manually validated the detection results of a statistically significant sample. Our next evaluation strategy involved assessing IDEAL on the sample dataset utilized to evaluate LAPD by comparing the detection results. In the following subsections, we provide details on these two evaluation activities, including numbers around the correctness of IDEAL and qualitative findings based on our manual analysis of source code.

Table 9.1: Summary of the linguistic anti-pattern detection rules IDEAL utilizes.

Id	Pattern	Detection Strategy
A.1	"Get" more than accessor	Impacted Identifiers: Method Names (excludes test methods) The name starts with 'get', the access specifier is public/protected, the name contains the name of an attribute, the return type is the same as the attribute type, and the body contains conditional statements
A.2	"Is" returns more than a Boolean	Impacted Identifiers: Method Names (excludes test methods) The name starts with a predicate/affirmation related term and the return type is not boolean
A.3	"Set" method returns	Impacted Identifiers: Method Names The name starts with 'set' and the return type is not void
A.4	Expecting but not getting single instance	Impacted Identifiers: Method Names (excludes test methods) The last term in the name is singular and the name does not contain terms that are a collection type and the return type is a collection
B.1	Not implemented condition	Impacted Identifiers: Method Names The name contains conditional related terms in the name or comment and body does not conditional statements
B.2	Validation method does not confirm	Impacted Identifiers: Method Names (excludes test methods) The name starts with a validation-related term, does not have a return type and does not throw an exception
B.3	"Get" method does not return	Impacted Identifiers: Method Names (excludes test methods) The name starts with a 'get' related term and the return type is void
B.4	Not answered question	Impacted Identifiers: Method Names (excludes test methods) The name starts with a predicate/affirmation related term and the return type is void
B.5	Transform method does not return	Impacted Identifiers: Method Names (excludes test methods) The name starts with or an inner term constains a transformation term and the return type is void
B.6	Expecting but not getting a collection	Impacted Identifiers: Method Names (excludes test methods) The name starts with a 'get' related term, the name contains a term that is either plural or a collection type and the return type is not a collection-based type
C.1	Method name and return type are opposite	Impacted Identifiers: Method Names (excludes test methods) An antonym relationship exists between terms in an identifiers name and data type
C.2	Method signature and comment are opposite	Impacted Identifiers: Method Names (excludes test methods) An antonym relationship exists between either terms in an identifiers name or data type and comments
D.1	Says one but contains many	Impacted Identifiers: Attributes, Method Variables and Parameters The last term in the name is singular and the data type is a collection
D.2	Name suggests Boolean but type does not	Impacted Identifiers: Attributes, Method Variables and Parameters The starting term should be predicate/affirmation related and the data type is not boolean
E.1	Says many but contains one	Impacted Identifiers: Attributes, Method Variables and Parameters The last term in the name is plural and the data type is not a collection
F.1	Attribute name and type are opposite	Impacted Identifiers: Attributes, Method Variables and Parameters An antonym relationship exists between terms in an identifiers name and data type
F.2	Attribute signature and comment are opposite	Impacted Identifiers: Attributes, Method Variables and Parameters An antonym relationship exists between either terms in an identifiers name or data type and comments
G.1	Name contains only special characters	Impacted Identifiers: Attributes, Method, Method Variables and Parameters The name of the identifier is composed of only non-alphanumeric characters
G.2	Redundant use of "test" in method name	Impacted Identifiers: Methods (excludes non-test methods) The name starts with the term 'test'

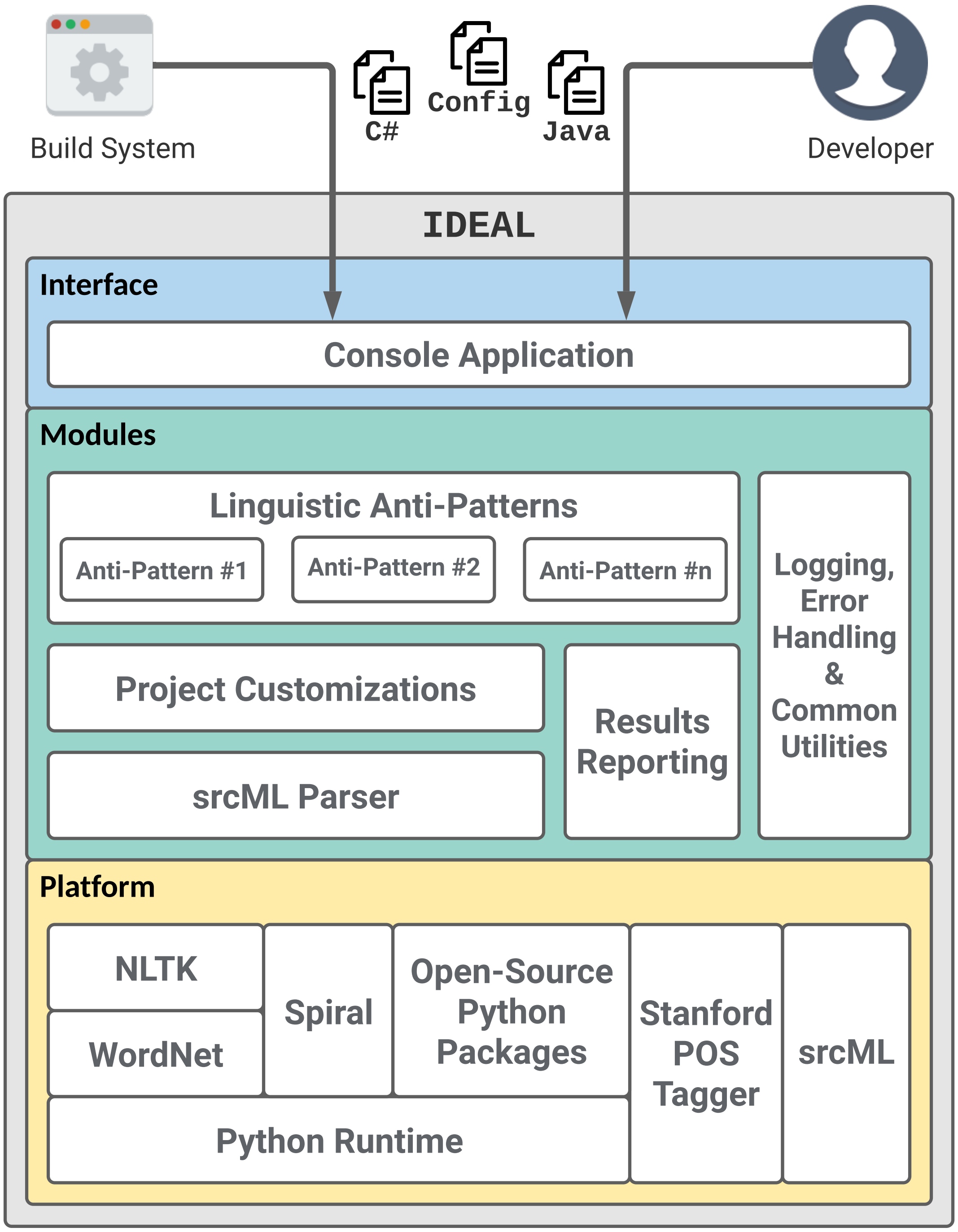


Figure 9.1: Conceptual architectural view of IDEAL.

Table 9.2: Summary of the systems in our evaluation process.

System	Language	Version	Release Date	Files Analyzed	Issues Detected
Retrofit	Java	2.9.0	May-2020	282	192
Jenkins	Java	2.293	May-2021	1,688	4,818
Shadowsocks	C#	4.4.0.0	Dec-2020	88	275
PowerShell	C#	7.1.3	Mar-2021	1,290	8,455

9.3.1 Evaluation on open-source systems

IDEAL can analyze systems implemented in any language supported by srcML. However, currently, it has only been evaluated using Java and C#. Thus, we selected two popular open-source systems for each of these programming languages. To this extent, Retrofit [2] and Jenkins [3] were the two Java systems, while Shadowsocks [4] and PowerShell [5] were the C# systems; Table 9.2 summarizes the release of each system that was part of our evaluation analysis.

For each of the four systems, we manually analyzed a random stratified statistically significant (i.e., confidence level of 95% and confidence interval of 10%) set of detected violations for each category. In total, we manually verified 2,019 instances of naming violations spread across the four systems. Table 9.3 provides a breakdown of the number of violation instances for each category. As part of the manual analysis process and to mitigate bias, we had discussions around specific violation instances that were subjective and, at times, referenced literature (grey and reviewed) to aid in the decision-making process. IDEAL reports an average precision of 75.27%, with 14 out of 19 violation types reporting a precision of over 50%. Though LAPD reports an average precision of 72%, we manually validate 1,267 more instances than LAPD. Furthermore, even though IDEAL supports customization per project (e.g., specifying custom collection data types and terms), our evaluation strategy did not utilize this feature in order to maintain consistency in violation detection across the four systems.

From Table 9.3, we observe that while IDEAL performs notably well in detecting all A.*, D.*, and E.* violations (precision score of over 80%). These are anti-patterns where the identifier either does or contains more than what is required. In most instances, IDEAL can accurately process the return/data type of the identifier to determine violations. However, there are also violations that are challenging for IDEAL to analyze and hence result in a large volume of false positives (e.g., C.2). Our manual analysis of these false-positive instances shows patterns that, in most cases, are causing IDEAL to report them as issues. First, since developers utilize custom data/return types for identifiers in their code, IDEAL fails in identifying their intended purpose. For instance, 'EnvVars' is a custom type created by a developer to hold a collection of specific items. The developer returns this type in a method called 'getEnvironmentVariables2'. Since IDEAL is unaware that 'EnvVars' is a collection-based type, it flags this as a violation since the method is supposed to return a collection (i.e., this get method name contains a plural term– 'Variables'). We are confident that once developers configure IDEAL to handle custom types, false positives, similar to this, will reduce. Our next observation is on how IDEAL analyzes lexical relationships between words; specifically, concerning antonyms (i.e., C.* and F.*). While IDEAL correctly recognizes antonyms, the context around how these terms are used, either in the identifier's name or comment, is not considered, resulting in false positives. Additionally, we also observe that naming habits/conventions also cause the emergence of antonyms. For instance, consider the method 'GetCompletionResult' with a return type called 'CompletionResult'. IDEAL determines that 'Get' and 'Result' are antonyms, which are lexically valid, but a false positive due to naming conventions. Similar to the last challenge, context around the use of transformation terms (i.e., B.5) and conditional terms (i.e., B.1) cause the reporting of a high volume of false positives. While IDEAL correctly detects these terms in the source code, how the developer utilizes the term in a name or comment is currently a challenge.

Finally, our manual review of the source code also allowed us to observe other poor naming/coding practices, which can be future linguistic anti-patterns. For example, the generic terms 'data' and 'result' are subjective. When used as part of an identifier's name, it is unknown if the identifier handles a single item or collection of items. Likewise, the use of the type 'var' (in C#) and 'object' also does not indicate the type of data the identifier handles. Ideally, to convey the purpose/behavior of the identifier correctly, developers need to be specific in naming identifiers and data types when possible.

9.3.2 Comparison with LAPD

In this part of our evaluation we compare the correctness of IDEAL with LAPD. To this extent, we analyze a sample of the source files that were utilized to evaluate the effectiveness of LAPD and compare the results. Since IDEAL implements the anti-patterns available in LAPD, it is essential to understand the areas where IDEAL under- and overperforms. In total, we analyzed 209 Java files and detected 294 violations. From this, both IDEAL and LAPD matched 199 true positive instances and 19 false positive instances. Furthermore, 47 instances identified as LAPD false positives were not detected by IDEAL, highlighting where IDEAL outperforms LAPD. Most of these instances were associated with C.2, D.1, and E.1. Finally, we also encounter instances where IDEAL does not detect LAPD true positives. While some of these issues are due to custom

Table 9.3: Summary of the detection correctness of IDEAL.

Id.	**Detected Instances**	**Validated Samples**	**True Positives**	**False Positives**	**Precision**
A.1	53	34	34	0	100.00%
A.2	45	37	37	0	100.00%
A.3	129	64	63	1	98.44%
A.4	341	127	102	25	80.31%
B.1	912	171	73	98	42.69%
B.2	446	166	165	1	99.40%
B.3	260	101	101	0	100.00%
B.4	18	16	5	11	31.25%
B.5	271	107	46	61	42.99%
B.6	827	159	128	31	80.50%
C.1	139	74	54	20	72.97%
C.2	294	112	13	99	11.61%
D.1	3,359	262	261	1	99.62%
D.2	83	53	53	0	100.00%
E.1	5,506	268	253	15	94.40%
F.1	38	32	19	13	59.38%
F.2	165	91	15	76	16.48%
G.1	1	1	1	0	100.00%
G.2	853	144	144	0	100.00%
Overall	13,740	2,019	1,567	452	75.27%

data types, we also encounter subjective instances, most of which (10 instances) fall under D.2.

Chapter 10

An Ensemble Approach for Annotating Source Code Identifiers with Part-of-speech Tags

The contents of this chapter are part of the study "***An Ensemble Approach for Annotating Source Code Identifiers with Part-of-speech Tags***" published in IEEE Transactions on Software Engineering [178].

10.1 Introduction

Program comprehension is a significant factor in the time it takes to develop and maintain software [121, 165]. Developers spend much more time reading code than they spend writing; 10 times more by some estimates [165]. Increased understanding of developer comprehension will lead to approaches that not only augment the ability of developers and program analysis tools to be productive, but also improve the accessibility of software development (e.g., by supporting programmers that prefer top-down or bottom-up comprehension styles [132, 221]) and help developers avoid stress stemming from code that is hard to understand. One of the primary ways a developer comprehends code is by reading identifier names, which make up on average about 70% of the characters found in a body of code [125]. Therefore, improving identifier naming practices can have a significant, positive impact on comprehension.

One challenge to studying identifiers is the difficulty in understanding how to map the meaning of natural language phrases to the behavior of the code. For example, when a developer names a method, the name should reflect the behavior of the method such that another developer can understand what the method does without the need to read the method body. Understanding this connection between name and behavior presents challenges for humans and tools; both of which use this relationship to comprehend, generate, or critique code. A second challenge lies in the natural language analysis techniques themselves, many of which are not trained to be applied to software [107]; introducing significant threats [150]. Addressing these problems is vital to improving the developer experience and augmenting tools which leverage natural language.

Analysis of identifier names can be done in many ways, including word frequency analysis [163] (e.g., ngrams) or semantic analysis using lexical ontologies like wordnet [168]. These are applied to a large number of problems, including rename refactoring analysis [101, 161, 185, 192], linguistic anti-patterns [103], identifier splitting [142], and part-of-speech tagging [137, 141, 180]. In this paper, we focus on part-of-speech tagging (POS); a technique whereby words in a sentence, or in an identifier in this case, are annotated based on the role they play within the context of the words surrounding them or based on their typical usage in the case where we are dealing with a single-word identifier. Part-of-speech tagging is one of the most popular methods for measuring the natural language semantics of identifier names and has been used in numerous other research [103, 113, 115, 137, 139, 145, 157, 183, 192]. Unfortunately, part-of-speech taggers for identifiers are still inaccurate [177, 180], making it difficult to trust their output.

The goal of this study is to discuss and present an ensemble tagging technique that improves the accuracy of part-of-speech taggers, and supports a larger variety of POS tags than other software engineering based POS taggers, specifically SWUM and POSSE [137, 141]. The ensemble approach uses machine-learning algorithms such as Decision Tree [171] and Random Forest [111], which are common in other software research tasks [91, 204] and have been used for part-of-speech tagging of standard English documents [126].

In addition to advancing the state-of-the-art of part-of-speech tagging, we also discuss where our approach is still weak; highlighting situations to which it may not sufficiently generalize due to potential limitations in our technique and our dataset. We answer the following Research Questions (RQs):

RQ1: How accurate is the ensemble part-of-speech tagger at the individual word level and how does this compare to the independent taggers? In prior research [177], we found that the output of three part-of-speech taggers complemented one another by applying them all to a manually-curated dataset of grammar patterns. This question will address just how much we can improve the accuracy of part-of-speech taggers on source code by combining the output of these taggers using machine learning.

RQ2: How accurate is the ensemble part-of-speech tagger at the identifier level and how does this compare to the independent taggers? In RQ1, we explore word-level accuracy.

In RQ2, we will look at how accurate our ensemble is when it must annotate the entire identifier correctly, since, as shown in prior work [177], even if a tagger has high accuracy on individual words, it may have low accuracy on full identifiers.

RQ3: What are the most frequently mis-used part-of-speech tags and grammar patterns? This question investigates whether there are patterns in the way our approach mis-annotates identifiers. We explore these cases and discuss what further information the ensemble requires in order to handle these cases properly. For example, in prior work [177], we found that implementation details have an effect on the correct tag sequence for certain identifiers and are thus required to properly tag these identifiers. In this question, we take a deeper look at this problem among others.

10.2 Methodology

In prior work [177], we constructed a dataset of 1,335 identifiers from 20 systems and manually annotated these identifiers with part-of-speech tags. For our evaluation of the ensemble tagger, we needed to create both a test set and a training set. We wanted the test set to contain identifiers from systems that were not present in the training set. Thus, we removed 5 systems and all corresponding identifiers from the original 20 system dataset and used these to create a test set. Since we wanted to maintain the same size as the original dataset, we collected additional identifiers from the remaining 15 systems so that the size of the dataset continued to be 1,335 identifiers. Thus, the training set used for our ensemble is derived from our original dataset [177] but is not the exact same.

Below, we explain our steps as if we collected and annotated the full 15 system dataset from scratch, as opposed to deriving it from prior work and expanding it, since these are all the steps used on all identifiers in our training and test sets. Explaining it this way simplifies the discussion. One thing to note is that we assess the quality of our model in multiple ways, meaning we have multiple test sets. We name these differently to ease the burden of reading. One set is called the ”unseen test set” which is made up of identifiers from systems that were not in the training set. The other test set(s) are constructed using 5-fold cross validation, which splits the 15 system training set into smaller train-test sets. We call this the ”5-fold test set”. We will stick to this terminology for clarity throughout this section.

Table 10.1: Distribution of Annotations in Training and Test Sets

Annotation	Training Set	Unseen Test Set
CJ	11	1
D	20	7
DT	13	5
N	1149	321
NM	1520	414
NPL	220	78
P	91	32
PRE	83	33
V	330	81
VM	12	3
Total	3449	977

10.2.1 Training Set Construction

We grouped identifiers into five categories: class names, function names, parameter names, attribute names (i.e., data members), and declaration-statement names. A declaration-statement name is a name belonging to a function-local or global variable. We use this terminology as it is consistent with srcML's terminology [119] for these variables and we used srcML to collect identifiers.

This dataset includes 1,335 identifiers which break down into 3,449 words (Table 10.1). The number was chosen by taking a sample from the total number of identifiers at 95 confidence level and 6 confidence interval from each of the five categories we support, meaning that we sampled 267 identifiers from each category (5*267=1,335). We collected our identifier set from 15 open-source systems. We chose these systems to vary in terms of size and programming language while also being mature and having their own development communities. We did this to make sure that the identifiers in these systems have been seen by multiple contributors and that the identifiers we collected are not biased toward a specific programming language. There are two reasons for choosing identifiers from multiple languages. 1) We want to know what patterns cross-cut between languages, such that most Java/C/C++ developers are familiar with and leverage these patterns.

Focusing on just one language might mean the patterns we find are not common to developers outside of the chosen language. 2) Many systems are written in more than one language, and it is important to understand how well part-of-speech tagging technologies will work on these systems. Thus, running our study on systems written in different programming languages helps us study part-of-speech tagger results in an environment leveraging multiple programming languages.

We provide the list of systems and their characteristics in Table 10.2. The systems we picked were 615 KLOC on average with a median of 476 KLOC, a min of 30 KLOC, and a max of 1,800 KLOC. Further, most of these systems have been in active development for the past ten years or more and all of them for five years or more. The younger systems in our set are popular, modern programs. For example, Swift is a well-known programming language supported by Apple, Telegram is a popular messaging app, and Jenkins is a popular development automation server. Because we are trying to train an ensemble part-of-speech tagger to work well on as much code as possible, our goal is not necessarily to include only high-quality identifier names, but to include names that are closely representative of the average name for open-source systems. Additionally, we remove identifiers that appear in test files, in part because they sometimes have specialized naming conventions (e.g., include the word ‘test’, ‘assert’, ‘should’, etc). This is supported by other research on test names [222, 225, 226]. We exclude test-related identifiers by ignoring annotated test files and directories; any directory, file, class, or function containing the word *test*. While it is possible that identifiers in test code have similar grammar patterns to identifiers outside of test code, it is also possible that they do not. We did not want to risk introducing divergent grammar patterns. We think it would be appropriate to create a separate dataset of test identifier grammar patterns.

To collect the 1,335 identifiers, we scanned each of our 15 systems using srcML [119] and collected both identifier names/types and the category that they fell into (e.g., class, function). Then, for each category, we randomly selected one identifier from each system using a round-robin algorithm (i.e., we picked a random identifier from system 1, then randomly selected an identifier from system 2, etc. until we hit 267). This ensured that we got either 17 or 18 identifiers from each system (267/15 = 17.8) per category and mitigates the threat of differing system size.

The dataset is balanced in terms of system (i.e., equal number of observations from each system) and in terms of code category (i.e., equal number of function names, parameter names, etc). However, the dataset is **not** balanced in terms of annotation. As shown in Table 10.1, there are a different number of observations for each annotation we support in our tag set. We do not balance it for 3 reasons: 1) the unbalanced nature of the dataset mirrors reality more accurately; some annotations are much less common than others in English. 2) Balancing it would cause our data

Table 10.2: Systems used to create training (unbolded) and unseen test (**bolded**) sets.

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

to be significantly different from the average distribution of identifier names [177]. 3) Because no automated tagger is 100% accurate, it would not be possible to automatically balance the dataset.

We did not expand abbreviations. The reason for this is that abbreviation expansion techniques are not widely available (e.g., cannot be easily integrated into different languages or frameworks, cannot be readily trained, are not fully or publicly implemented) and still not very accurate [176]. Therefore, a realistic worst-case scenario for developers and researchers is that no abbreviation-expansion technique is available to use; their part-of-speech taggers must work in this worst-case scenario. We also tried not to split domain-term abbreviations (e.g., some splitters will make IPV4 into IPV 4; we corrected this back to IPV4). We did this because some taggers may recognize these domain terms. Furthermore, we are also of the view that these terms should be recognized and appropriately tagged in their abbreviated (i.e., their most common) form.

Some verb forms are used as either adjectives or verbs. Stanford tagger is the only tagger that recognizes derivative verb forms such as past-tense or modal. Thus, it is the only one we need to configure. In prior work [177] that verbs are being used as adjectives or verbs. In short, Stanford's accuracy increases when we assume that verb conjugations are adjectives in every context except function names. For function names, it is better to assume that its verb annotations are verbs. Thus, our training set reflects this reality.

Finally, when we apply the Stanford tagger to function names, we append the letter *I* to the beginning of the name. This is a known technique– the Stanford+I technique, used to help Stanford tag identifiers that represent actions more accurately. It was used in previous studies applying part-of-speech tags to method identifier names [86, 106, 177, 180] to increase Stanford's accuracy and confirmed to increase Stanford's accuracy on function names [177].

10.2.2 Unseen Systems Test Set

Our test set is made up of 384 identifiers that break down into 977 words (Table 10.1) grouped by the same five categories used for the training set. It is constructed from identifiers contained in 5 systems that were removed from the original dataset [177], as explained at the beginning of Section 10.2 and shown **bolded** in Table 10.2. We based the number, 384, on a sample at 95 confidence level and 5 confidence interval. The population from which the sample was derived is the full set of identifiers across all categories. The size and breakdown of the full population, 2,743,252 identifiers, is found in Table 6.1. Given this sample size, we collected 76 or 77 (384/5 = 76.8) identifiers from each category. These were balanced for category as well as system (i.e., an equal number of identifiers from each system). Like the training set, we manually annotated these

Table 10.3: Part-of-speech categories in dataset and supported by ensemble

Abbreviation	**Expanded Form**	**Examples**
N	noun	Disneyland, shoe, faucet, mother
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, above
NPL	noun plural	Streets, cities, cars, people, lists
NM	noun modifier (adjectives, noun-adjuncts)	red, cold, hot, scary, beautiful, small
V	verb	Run, jump, spin
VM	verb modifier (adverb)	Very, loudly, seriously, impatiently
PR	pronoun	she, he, her, him, it,we,they,them
D	digit	1, 2, 10, 4.12, 0xAF
PRE	preamble	Gimp, GLEW, GL, G, p, m, b

identifiers with part-of-speech tags, with each annotator taking a set to annotate on their own and cross-validated by swapping sets to confirm (i.e., agree or disagree) that the manual annotation is correct. The annotators came to a complete agreement on each identifier.

There are multiple versions of this dataset. All of them have the same identifiers, but the annotations change based on which configuration was used to generate the set. The configurations come in pairs. One pair is *normalized* or *conjugated* and the other pair is *augmented* or *plain*. *Normalized* datasets are those which convert all verb conjugations detected by standford to standard verbs (V in Table 10.3). *Conjugated* datasets are the opposite; they used all Stanford's verb conjugations. *Augmented* datasets remove all annotations that have a frequency less than 25 and replaced them with an OTHER category. We use this to study whether rarely-seen part-of-speech tags have a negative effect on the overall quality of the tagger. *Plain* datasets include all annotations shown in Table 10.3.

10.2.3 5-fold test set

In addition to the unseen test set, we use k-fold cross validation to help us understand the generality of our ensemble tagging model. Typically, when using k-fold cross validation, prior researchers choose k as either 5 or 10. In this work, we choose 5 since it is most appropriate considering the distribution of annotations and the size of our dataset. The 5-fold test set is constructed from the training set of 1,335 identifiers. Effectively, the training set is split into five smaller sets. 30% of these is chosen as a testing set and the other 70% are used as training. Then, after training on four and testing on one, following typical 5-fold cross validation procedure, other sets are chosen as the test sets and the rest (now including some data that was just used as testing) are used to train. At each train-test step, we collect metrics about the effectiveness of our model as discussed in the next subsection.

10.2.4 Measuring Model Quality

We measure the quality of our ensemble using typical metrics for categorization problems. That is, we use Accuracy, Precision, Recall, and F1 Score. In addition to the metrics above, we also report balanced accuracy in our 5-fold results, which is similar to accuracy except we calculate the average proportion of correct predictions for each individual annotation (i.e., N, NM, CJ, etc.) first and then divide by the number of annotations. Balanced accuracy helps when dealing with unbalanced datasets by giving more weight to annotations with lower frequency in the dataset.

Table 10.4: Optimal parameter values for the classification algorithms.

Algorithm	**Parameter**	**Value**
Random Forest	max_depth	83
	n_estimators	250
	criterion	gini
	bootstrap	True
Decision Tree	criterion	entropy
	max_depth	9

10.2.5 Choosing and training machine learning approaches

For the evaluation of our ensemble, we chose to use Random Forest and Decision Tree as our primary machine learning approaches. Initially, we considered Support Vector Classification, Logistic Regression, K-Nearest Neighbors, and Multinomial Naive Bayes. However, our preliminary analysis shows that Random Forest and Decision Tree outperforms the other classifiers in terms of our model quality metrics. Hence, we focus on evaluating the quality of our approach to using these two algorithms. To build our optimized model, we first split the dataset into a training and test set. The training set contains 70% of the observations, while the remaining 30% of the observations were part of the test set as validation data. To ensure that we are constructing an optimized model, we perform a hyperparameter optimization process. The purpose of this process is to evaluate a series of parameter values associated with the model to determine the set of values that result in the best performance of the model [87]. Our hyperparameter tuning process involved an exhaustive grid search [123] and 5-fold cross-validation on the training dataset. Grid search utilizes a brute force technique to evaluate all combinations of hyperparameters to obtain the best performance. Provided in Table 10.4, are the optimal hyperparameter values for the classification algorithms in our study.

10.2.6 Dataset preparation and Features

To prepare the datasets for annotation by the ensemble, we run SWUM [141], POSSE [137], and Stanford [217] on each identifier to obtain their individual annotations. We provide any information required by the three taggers above (e.g., SWUM requires function signatures). Since the dataset is pre-split from prior work [177], we do not have to worry about splitting. In addition, as stated prior, the correct annotation is already provided. After we have run these three taggers on the data, we vectorize the data by splitting all identifiers into their individual words along with the part-of-speech provided to them by SWUM, POSSE, Stanford, and the human annotators. In addition, we collect several data characteristics to serve as features to help the ensemble correctly annotate the data. We explain these characteristics now.

Machine learning algorithms use characteristics of the data they are trained on to learn the nuances of that data such that they are able to use these characteristics to categorize unseen data. These characteristics are typically called Features. Our ensemble uses several features to annotate (i.e., categorize) identifiers with part of speech tags. The features that we considered for our model are based on empirical results from a prior study we performed on the grammar patterns latent in source code identifiers [177]. To summarize, we found that certain annotations are heavily correlated with 1) words that appear in certain positions. For example, nouns appear at the end of an identifier, noun-adjuncts appear at the beginning or middle; 2) with the type of an identifier. For example, verbs are more common in boolean-type identifiers; and 3) certain contexts. For example, verb phrases are more common in function names. We also noticed that certain taggers are better at recognizing certain annotations than others. For example, SWUM is great at recognizing noun-adjuncts, Stanford is great at recognizing conjunctions and prepositions. Therefore, we chose features that will help our ensemble take advantage of these patterns. Most of these features are also very easy to obtain using static analysis, making them very accessible in different environments and, thus, helping guarantee ease of integrating our approach into another applications. The features that we considered for our model are as follows:

1. **Word** - The word itself.
2. **Data Type** - the type (or return type) of an identifier (or function identifier).
3. **SWUM annotation** - The annotation that the SWUM POS tagger applied to a given word.
4. **POSSE annotation** - The annotation that the POSSE POS tagger applied to a given word.
5. **Stanford annotation** - The annotation that the Stanford POS tagger applied to a given word.
6. **Position** - The position of a given word within its original identifier. For example, given an identifier: *GetXMLReaderHandler*, *Get* is in position 1, *XML* is in position 2, *Reader* is in position 3 and *Handler* is in position 4.
7. **Identifier size** - The length, in words, of the identifier of which the word was originally part.

8. **Normalized position** - We normalized the position metric described above such that the first word in the identifier is in position 1, all middle words are in position 2, and the last word is in position 3. For example, given an identifier: *GetXMLReaderHandler*, *Get* is in position 1, *XML* is in position 2, *Reader* is in position 2 and *Handler* is in position 3. The reason for this feature is to mitigate the sometimes-negative effect of very long identifiers.

9. **Context** - The dataset contains five categories of identifier name: function, parameter, attribute, declaration, and class. We provide the category to which the given identifier belongs as one of the features to allow the ensemble to learn patterns that are more pervasive for certain identifier types versus others. For example, function identifiers contain verbs at a higher rate [137, 141, 145, 177] than other types of identifiers.

We tested all of these features using 5-fold cross validation and the metrics Described in Section 10.2.4. Specifically, we were trying to determine what set of features maximized F1, Accuracy, and Balanced accuracy. To do this, we used 2 techniques: **Drop-column feature importance** and **permutation importance**. Drop-column feature importance can by calculated by creating a power set (i.e., all subsets) of the full set of features and then retraining your model with each subset. In this way, we can consider every possible subset of features for our feature set and determine which subset gives us the best performance with respect to F1, Accuracy, and Balanced Accuracy. While performing Drop-column feature importance, we also performed permutation importance. Permutation importance is done after a model has been fitted. It is defined as the decrease in a model score (i.e., our metrics) when a single feature's value is randomly shuffled. In essence, it measures how our metrics change when a feature is shuffled. Thus, for each subset of features in our ensemble, we also measure permutation importance.

Since there are a lot of subsets (i.e., power set of our 9 features is $2^9 = 512$), we only present data about the best feature set: **SWUM**, **POSSE**, and **Stanford** annotations, **Normalized position**, and **Context**. In addition, we present the permutation importances for these features in Tables 10.5 and 10.6. These tables correspond to permutation importances for our best Random-Forest-based ensemble and our best Decision-Tree-based ensemble. In each table, you can see how each of the best features influenced F1, Balanced Accuracy, and Accuracy. Since we used 5-fold validation, there are 5 values in each row followed by an average of those values. *A higher number means a feature is more important.* In general, out of our three taggers, SWUM had the highest influence on F1 and Accuracy, while Stanford had the highest influence on Balanced Accuracy. Of the non-tagger features (i.e., Normalized Position and Context), Normalized Position had the highest influence on all three metrics.

Finally, the features that were removed: Word, data type, position, and identifier size degraded our model performance. Our prior work [177] gives us some insight as to why this might be. Starting with *position* and *max position*, we found that verb and noun phrases tended to begin with a particular annotation; a verb or noun modifier respectively. They also ended with a specific annotation: a Noun (i.e., a head-noun). Between the starting verb/noun modifier and the ending head-noun are a sequence of Noun Modifiers. Notice that this correlates to a *beginning*, *middle*, *end* structure where the first word has a specific tag, all the middle words have the same tag, and the final word has a specific tag. Position and Max position confuse the ensemble because identifiers have varying lengths. Normalized position categorizes position as beginning, middle, or end. Thus, it improves the performance of the ensemble whereas position and max position hurt the performance. Word and data type can help the model recognize certain words and their correlation to different tags. However, on unseen data this may cause the ensemble to become confused because it sees words that it has not seen before. Thus, the ensemble will tag common words more accurately with these features turned on but uncommon/unseen words less accurately.

10.3 Evaluation Setup

The dataset described in Section 10.2 has several configurations that we use during evaluation. These configurations are as follows:

1. The type of machine learning approach used; either Decision Tree or Random forest. They have the codes DT and RF respectively.
2. The version of the dataset being used; either the plain dataset or the augmented dataset. These have the codes P and A respectively.
3. Whether or not the Stanford data within the dataset is using verb conjugations or is normalized. These have the codes C and N respectively.

To determine which configuration you are looking at when reading our results, look at the code present in each table. For example, if some data in a table has the code RFCP, then it used **random forest** with **conjugated** stanford identifiers and the **plain** dataset.

Table 10.5: Decision Tree feature importances for best features

Feature Set	F1 Weighted Importances					Average
SWUM	0.26	0.26	0.25	0.26	0.26	0.26
POSSE	0.15	0.14	0.15	0.15	0.15	0.15
Stanford	0.23	0.23	0.23	0.23	0.23	0.23
Context	0.02	0.02	0.02	0.02	0.02	0.02
Nomalized Position	0.13	0.13	0.12	0.13	0.13	0.13
Feature Set	Balanced Accuracy Importances					Average
SWUM	0.29	0.27	0.25	0.28	0.27	0.27
POSSE	0.21	0.21	0.21	0.24	0.21	0.22
Stanford	0.51	0.51	0.49	0.53	0.51	0.51
Context	0.05	0.07	0.05	0.07	0.07	0.06
Nomalized Position	0.10	0.10	0.08	0.10	0.08	0.09
Feature Set	Accuracy Importances					Average
SWUM	0.26	0.26	0.26	0.26	0.26	0.26
POSSE	0.14	0.13	0.13	0.13	0.13	0.13
Stanford	0.22	0.21	0.20	0.20	0.21	0.21
Context	0.02	0.02	0.02	0.02	0.02	0.02
Nomalized Position	0.13	0.13	0.14	0.13	0.14	0.13

Table 10.6: Random Forest feature importances for best features

Feature Set	**F1 Weighted Importances**					**Average**
SWUM	0.22	0.22	0.21	0.21	0.22	0.21
POSSE	0.14	0.15	0.15	0.14	0.15	0.14
Stanford	0.21	0.21	0.22	0.21	0.21	0.21
Context	0.03	0.03	0.03	0.03	0.03	0.03
Nomalized Position	0.17	0.16	0.15	0.16	0.15	0.16
Feature Set	**Balanced Accuracy Importances**					**Average**
SWUM	0.22	0.22	0.21	0.26	0.25	0.23
POSSE	0.23	0.23	0.20	0.23	0.21	0.22
Stanford	0.47	0.47	0.50	0.50	0.49	0.48
Context	0.05	0.04	0.07	0.06	0.06	0.06
Nomalized Position	0.15	0.12	0.15	0.19	0.21	0.17
Feature Set	**Accuracy Importances**					**Average**
SWUM	0.23	0.22	0.21	0.22	0.22	0.22
POSSE	0.14	0.13	0.13	0.13	0.13	0.13
Stanford	0.19	0.18	0.19	0.19	0.19	0.19
Context	0.03	0.03	0.03	0.02	0.02	0.03
Nomalized Position	0.16	0.16	0.15	0.16	0.16	0.16

Table 10.7: Five-fold validation results for each machine learning approach and configuration using the augmented dataset

	DTNA					**Average**	**RFNA**					**Average**
Accuracy	0.80	0.84	0.79	0.84	0.82	**0.82**	0.81	0.83	0.79	0.84	0.84	**0.82**
Balanced Accuracy	0.54	0.68	0.65	0.66	0.59	0.62	0.57	0.66	0.71	0.65	0.75	**0.67**
Weighted F1	0.79	0.83	0.79	0.83	0.81	0.81	0.80	0.82	0.79	0.84	0.84	**0.82**
Weighted Precision	0.80	0.82	0.80	0.83	0.82	0.81	0.81	0.82	0.80	0.84	0.84	**0.82**
Weighted Recall	0.80	0.84	0.79	0.84	0.82	**0.82**	0.81	0.83	0.79	0.84	0.84	**0.82**
	DTCA					**Average**	**RFCA**					**Average**
Accuracy	0.82	0.85	0.81	0.84	0.84	0.83	0.84	0.86	0.81	0.85	0.86	**0.84**
Balanced Accuracy	0.51	0.62	0.55	0.53	0.58	0.56	0.52	0.69	0.62	0.48	0.65	**0.59**
Weighted F1	0.82	0.84	0.80	0.82	0.82	0.82	0.83	0.86	0.80	0.84	0.86	**0.84**
Weighted Precision	0.83	0.84	0.81	0.82	0.82	0.82	0.83	0.87	0.80	0.85	0.86	**0.84**
Weighted Recall	0.82	0.85	0.81	0.84	0.84	0.83	0.84	0.86	0.81	0.85	0.86	**0.84**

Table 10.8: Five-fold validation results for each machine learning approach and configuration using the plain dataset

			DTNP			Average			RFNP			Average
Accuracy	0.81	0.81	0.86	0.82	0.82	0.82	0.84	0.82	0.87	0.82	0.85	**0.84**
Balanced Accuracy	0.54	0.59	0.71	0.75	0.53	0.62	0.61	0.60	0.76	0.74	0.60	**0.66**
Weighted F1	0.79	0.80	0.86	0.81	0.80	0.81	0.82	0.82	0.87	0.81	0.84	**0.83**
Weighted Precision	0.80	0.80	0.87	0.82	0.79	0.82	0.82	0.82	0.87	0.81	0.85	**0.83**
Weighted Recall	0.81	0.81	0.86	0.82	0.82	0.82	0.84	0.82	0.87	0.82	0.85	**0.84**
			DTCP			**Average**			**RFCP**			**Average**
Accuracy	0.85	0.82	0.83	0.84	0.85	**0.84**	0.86	0.83	0.83	0.85	0.85	**0.84**
Balanced Accuracy	0.58	0.67	0.73	0.60	0.63	**0.64**	0.51	0.59	0.54	0.50	0.63	0.55
Weighted F1	0.85	0.82	0.83	0.84	0.84	0.83	0.86	0.82	0.82	0.84	0.84	**0.84**
Weighted Precision	0.85	0.81	0.83	0.83	0.83	**0.83**	0.86	0.82	0.82	0.83	0.83	**0.83**
Weighted Recall	0.85	0.82	0.83	0.84	0.85	**0.84**	0.86	0.83	0.83	0.85	0.85	**0.84**

Table 10.9: Per-annotation and Overall Accuracy of Ensemble Tagger on Augmented Dataset

			DTNA				RFNA				DTCA				RFCA		
Annotation	Total	Precision	Recall	F1	Total	Precision	Recall	F1	Total	Precision	Recall	F1	Total	Precision	Recall	F1	Total
N	322	0.87	0.89	**0.88**	329	0.86	0.89	**0.88**	331	0.84	0.89	0.87	340	0.87	0.90	**0.88**	332
NM	415	0.85	0.92	**0.89**	448	0.85	0.92	**0.89**	446	0.87	0.91	**0.89**	435	0.86	0.92	**0.89**	447
NPL	78	0.90	0.68	0.77	59	0.93	0.67	0.78	56	0.89	0.72	0.79	63	0.96	0.69	**0.81**	56
OTHER	16	0.69	0.56	0.62	13	0.73	0.69	**0.71**	15	0.56	0.31	0.40	9	0.50	0.25	0.33	8
P	32	0.70	0.81	0.75	37	0.71	0.84	0.77	38	0.68	0.78	0.72	37	0.72	0.88	**0.79**	39
PRE	33	0.64	0.21	0.32	11	0.64	0.21	0.32	11	0.78	0.21	0.33	9	0.77	0.30	**0.43**	13
V	81	0.79	0.78	0.78	80	0.79	0.78	0.78	80	0.79	0.81	0.80	84	0.80	0.81	**0.81**	82
Accuracy			**0.84**				**0.85**				**0.84**				**0.85**		

Table 10.10: Per-annotation and Overall Accuracy of Ensemble Tagger on Plain Dataset

			DTNP				RFNP				DTCP				RFCP		
Annotation	Total	Precision	Recall	F1	Total	Precision	Recall	F1	Total	Precision	Recall	F1	Total	Precision	Recall	F1	Total
CJ	1	0.50	1.00	0.67	2	1.00	1.00	**1.00**	1	1.00	1.00	**1.00**	1	1.00	1.00	**1.00**	1
D	7	0.88	1.00	**0.93**	8	0.88	1.00	**0.93**	8	0.88	1.00	**0.93**	8	0.88	1.00	**0.93**	8
DT	5	1.00	0.20	0.33	1	1.00	0.40	0.57	2	1.00	0.60	**0.75**	3	1.00	0.60	**0.75**	3
N	322	0.86	0.90	0.88	338	0.87	0.89	0.88	328	0.85	0.90	0.88	340	0.88	0.89	**0.89**	327
NM	415	0.85	0.93	**0.89**	452	0.85	0.93	**0.89**	453	0.88	0.91	**0.89**	430	0.87	0.92	**0.89**	438
NPL	78	0.98	0.65	0.78	52	0.93	0.67	0.78	56	0.90	0.72	**0.80**	62	0.93	0.69	0.79	58
P	32	0.65	0.81	0.72	40	0.72	0.88	**0.79**	39	0.68	0.84	0.75	40	0.70	0.88	0.78	40
PRE	33	0.70	0.21	0.33	10	0.64	0.21	0.32	11	0.78	0.21	0.33	9	0.77	0.30	**0.43**	13
V	81	0.85	0.77	0.81	73	0.81	0.77	0.78	77	0.80	0.81	**0.81**	82	0.79	0.83	0.81	85
VM	3	0.00	0.00	0.00	1	0.50	0.33	0.40	2	0.50	0.33	0.40	2	0.50	0.67	**0.57**	4
Accuracy			**0.85**				**0.85**				**0.85**				**0.86**		

Table 10.11: Per-annotation and Overall Accuracy of the Independent Taggers on Plain Dataset - N/A = annotation not supported by tagger

		SWUM				**POSSE**				**Stanford**			
Annotation	Total	Precision	Recall	F1	Total	Precision	Recall	F1	Total	Precision	Recall	F1	Total
CJ	1	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	**1.00**	**1.00**	**1.00**	1
D	7	0.33	0.14	0.20	3	N/A	N/A	N/A	N/A	**0.78**	**1.00**	**0.88**	9
DT	5	0.50	**0.60**	0.55	6	N/A	N/A	N/A	N/A	**1.00**	0.40	**0.57**	2
N	322	**0.74**	0.89	**0.81**	384	0.42	0.90	0.57	692	0.47	**0.93**	0.62	647
NM	415	0.78	**0.94**	**0.85**	500	0.82	0.21	0.33	106	**0.83**	0.09	0.17	47
NPL	78	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	**0.86**	**0.73**	**0.79**	66
P	32	**0.88**	0.44	0.58	16	0.61	0.63	0.62	33	0.58	**0.91**	**0.71**	50
PRE	33	**0.50**	**0.03**	**0.06**	2	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
V	81	**0.89**	0.72	**0.79**	65	0.77	0.75	0.76	79	0.51	**0.90**	0.65	44
VM	3	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	**0.30**	**1.00**	**0.46**	10
Accuracy		**0.77**				**0.47**				**0.52**			

Table 10.12: Accuracy of independent taggers and best two ensemble taggers in different contexts at the word-level

	Total	DTCP	RFCP	SWUM	POSSE	Stanford
Attribute	194	0.82	**0.83**	0.72	0.42	0.45
Class	200	0.87	**0.89**	0.84	0.43	0.40
Declaration	184	0.83	**0.84**	0.79	0.48	0.45
Function	231	**0.85**	**0.85**	0.74	0.55	0.75
Parameter	168	**0.90**	0.89	0.77	0.45	0.53
Overall	977	0.86	0.86	0.77	0.47	0.52

Table 10.13: Average Accuracy at the identifier-level

Category	**Total**	**DTNA**	**RFNA**	**DTCA**	**RFCA**	**DTNP**	**RFNP**	**DTCP**	**RFCP**
Attribute	76	0.70	0.70	**0.72**	**0.72**	0.68	0.70	**0.72**	**0.72**
Class	77	0.78	0.78	0.77	**0.82**	0.77	0.78	0.77	**0.82**
Declaration	77	0.66	0.71	0.66	0.73	**0.74**	0.71	0.69	0.71
Function	77	0.62	0.62	0.68	0.65	0.64	0.64	**0.71**	0.69
Parameter	77	0.79	0.79	0.78	0.79	**0.81**	**0.81**	**0.81**	0.79
Overall	**384**	0.71	0.72	0.72	0.74	0.73	0.73	0.74	**0.75**

Table 10.14: Average accuracy at the identifier-level for state-of-the-art POS taggers

Category	**Total**	**SWUM**	**POSSE**	**Stanford**
Attribute	76	0.54	0.17	0.17
Class	77	0.61	0.13	0.19
Declaration	77	0.65	0.27	0.22
Function	77	0.48	0.30	0.29
Parameter	77	0.61	0.14	0.32
Overall	**384**	0.58	0.20	0.24

10.4 Experimental Results

10.4.1 RQ1: How accurate is the ensemble part-of-speech tagger at the individual word level and how does this compare to the independent taggers?

We evaluate accuracy in two ways. The first way is by running 5-fold cross validation using the manually-curated set of 1,335 identifiers. The second way is by running our model on an unseen test set of 384 identifiers. We will split our discussion of RQ1 results into these individual evaluations.

5-fold test results

The results from this evaluation are found in Table 10.7 and Table 10.8. These tables give the 5-fold results for five different metrics: accuracy, balanced accuracy, weighted f1, weighted precision, and weighted recall. We ran this 5-fold evaluation on 8 different configurations of our ensemble tagger. Overall, our results indicate that Random Forest gives the best results regardless of configuration; achieving the highest average in all five metrics used to gauge the quality of our ensemble when compared to decision tree. There is one exception, which is DTCP vs RFCP in Table 10.8. DTCP achieves the same averages compared with RFCP except with respect to balanced accuracy and weighted f1, where DTCP has better balanced accuracy and RFCP has a better weighted f1. If we look across both tables, then the best configurations that maximize all quality metrics are: DTCP, RFCP, RFNP, and RFCA. Out of those configurations, we would advise that the best configuration is DTCP or RFCP. The reason for this is that these configurations 1) are the most accurate on average; 2) use Stanford's conjugations instead of normalizing them away, meaning that they require less dataset preparation; and 3) they use the plain dataset, meaning that they are operating on the full tagset instead of grouping low-frequency annotations together under the OTHER category. This allows them to provide these annotations when they are used to tag identifiers.

Unseen test set results

The results from this evaluation are found in Table 10.9 and Table 10.10. Each table shows the precision, recall, f1, and accuracy of each ensemble configuration on each individual annotation supported by the ensemble. Table 10.9 has fewer annotations since less-frequent annotations are grouped under the OTHER category. Whereas Table 10.10 contains all annotations supported in our tagset even if they were very infrequent in the dataset. In addition, these tables show the overall accuracy for each ensemble tagger configuration on the unseen test set. This accuracy is obtained by measuring how many words the ensemble tagger annotated correctly when compared to the manual annotations. One thing to note about these results is that the *total* column on the leftmost side of each table is the total of each annotations in the manually-annotated (i.e., gold) set. Therefore, each configuration would have a different distribution since they each incorrectly annotated some words.

The overall results agree with our 5-fold results. That is, the best taggers tend to be random forest based ensembles. Again, with the exception of DTCP, which is competitive with the other random forest ensembles. RFCP does marginally better than DTCP on the unseen test set, achieving an accuracy of .86 versus DTCP's .85. In addition, since they are trained on the plain dataset, they could be used to annotate tags that are less frequent in production code: DT, CJ, VM, and D; each of which were grouped into the OTHER category in the dataset that RFCA was trained on. It is also notable that the most accurate ensemble configurations were trained on the plain dataset, indicating that the greater tag granularity helped improve the ensemble's output. We come to the same conclusion in this analysis as we did in the 5-fold analysis; DTCP and RFCP are the best ensemble for all of the same advantages explained above alongside retaining the best average values on our metrics. However, RFCP is marginally the better of the two based on our results. One of the primary reasons we still include DTCP as a competitive alternative to RFCP is that DTCP is faster and would scale better for large datasets.

Comparison with independent taggers

Table 10.11 shows the accuracy of the independent taggers at the word-level. Comparing the data in this table to Table 10.10, both DTCP and RFCP outperform or match the other taggers in every individual category. In addition, from Table 10.12, we see that DTCP and RFCP maintain their performance advantage at the context level as well.

Discussion of Feature Importance

In prior work [177], we noticed that the individual taggers had strengths and weaknesses that complemented one another. Specifically, Stanford was able to annotate Conjunctions, Digits, Determiners, Noun Plurals, Prepositions, and Verb Modifiers with high accuracy. Meanwhile, SWUM and POSSE tended to outperform Stanford in annotating Noun Modifiers and Verbs. Thus, Stanford's higher balanced accuracy makes perfect sense; it is very complementary to SWUM and POSSE. In addition, we also noticed that word position is important to annotating certain tags, such as Noun Modifiers. This is because, in a noun phrase, the leftmost words tend to be Noun Modifiers while the right-most word is a Noun (i.e., specifically, a head-noun). Another example is that Verbs tend to be in the first position in a function name.

Providing the normalized position helps the ensemble learn these patterns. Normalized position tends to be more effective at this than plain position because normalized position identifies the beginning, middle, and end of an identifier specifically. In contrast, raw position confuses the ensemble since identifiers can be of varying length, making it difficult to identify where the middle and end of an identifier are. Context is, surprisingly, not as important as normalized position. However, it is still part of the best feature set, meaning that it performs better than the subset of features that excludes context but includes normalized position. Thus, knowing whether an identifier is a function name, parameter, etc is still important for annotating with part-of-speech using our approach.

In summary, we have used two different approaches to evaluate our ensemble tagger. In addition, each of these approaches evaluated the ensemble using a set of unseen data to ensure that the ensemble is as general as possible; that the results from its evaluation will translate well to other unseen situations. Based on our data, DTCP and RFCP are equivalent in terms of average performance and have some minor differences between them when we look at which annotations they are most effective on. There is one advantage that DTCP has over RFCP that may be worth mentioning: it is faster. Since decision trees tend to be simpler models than random forests, DTCP generally annotates more rapidly than RFCP.

10.4.2 RQ2: How accurate is the ensemble part-of-speech tagger at the identifier level and how does this compare to the independent taggers?

In RQ1, we explore the accuracy of our ensemble tagger at the level of individual words. That is, we want to know how many words it correctly annotates in our dataset. However, the number of correctly annotated words does not give us the full picture. Since most identifiers in the code are made up of multiple words, it is also important to understand how accurate our ensemble tagger is on full identifiers. For this reason, we took the unseen test set and analyzed how effective our ensemble was on full identifier names.

The results of this analysis are given in Table 10.13 and Table 10.14. Table 10.13 shows the accuracy of our individual ensemble configurations, broken down by the five categories we used in our training set: attribute, class, declaration, function, and parameter. In addition, it shows the total number of each type of identifier in the dataset. Table 10.14 shows the individual accuracy of the three state-of-the-art part-of-speech taggers used to construct our ensemble. The numbers here are lower than in the prior tables from RQ1 because we are measuring full identifier names; if even a single word in the identifier name is mis-annotated, then we consider it incorrect.

Our results show that the overall accuracy of our ensemble on full identifier names is unsurprisingly lower than on individual words; about 11-13% lower in general. However, the ensemble still performs better than its closest competition according to both prior work [177] and our own analysis shown in Table 10.14, where we show the accuracy of individual part-of-speech taggers on the same unseen test set on which we ran our ensemble. Comparing the performance of the ensemble and the individual part-of-speech taggers, we can see that the best configuration of our ensemble (i.e., RFCP) outperforms the best tagger, SWUM, by around +17% on average while it outperforms POSSE and Stanford by 55% and 51% respectively.

In summary, these results are promising, but not surprising. Our ensemble is trained using the output of these approaches, so we would expect that it is able to learn their mistakes and produce output at a higher accuracy than its constituent taggers. We have shown that we can use an ensemble approach to improve upon part-of-speech tagging approaches on source code identifiers at both the individual word level and the full identifier level.

Table 10.15: Top 5 most frequently mis-annotated grammar patterns for each ensemble configuration

DTCA				DTCP			
Grammar Pattern	**# Incorrect**	**Actual**	**Proportion**	**Grammar Pattern**	**# Incorrect**	**Actual**	**Proportion**
NM NM NM NM N	6	8	0.75	NM NM NM NM N	6	8	0.75
NM NM NM N	24	38	0.63	NM NM NM N	25	38	0.66
PRE N	5	9	0.56	PRE N	5	9	0.56
V NM NM N	9	22	0.41	V NM NM N	9	22	0.41
V NM NPL	4	12	0.33	NM	2	6	0.33
RFCA				**RFCP**			
Grammar Pattern	**# Incorrect**	**Actual**	**Proportion**	**Grammar Pattern**	**# Incorrect**	**Actual**	**Proportion**
PRE N	6	9	0.67	PRE N	6	9	0.67
NM NM NM NM N	5	8	0.63	NM NM NM N	21	38	0.55
NM NM NM N	20	38	0.53	NM NM NM NM N	4	8	0.50
NM NM NM NPL	2	4	0.50	V NM NM N	9	22	0.41
V NM NM N	9	22	0.41	PRE NM N	4	12	0.33
DTNA				**DTNP**			
Grammar Pattern	**# Incorrect**	**Actual**	**Proportion**	**Grammar Pattern**	**# Incorrect**	**Actual**	**Proportion**
NM NM NM NM N	7	8	0.88	NM NM NM NM N	7	8	0.88
NM NM NM N	29	38	0.76	NM NM NM N	31	38	0.82
PRE N	6	9	0.67	PRE N	5	9	0.56
V NM NM N	10	22	0.45	V NM NM N	9	22	0.41
PRE NM N	4	12	0.33	V N P N	2	5	0.40
RFNA				**RFNP**			
Grammar Pattern	**# Incorrect**	**Actual**	**Proportion**	**Grammar Pattern**	**# Incorrect**	**Actual**	**Proportion**
NM NM NM NM N	6	8	0.75	NM NM NM NM N	7	8	0.88
NM NM NM N	28	38	0.74	NM NM NM N	29	38	0.76
PRE N	6	9	0.67	PRE N	6	9	0.67
V NM NM N	9	22	0.41	V NM NM N	9	22	0.41
PRE NM N	4	12	0.33	PRE NM N	4	12	0.33

10.4.3 RQ3: What are the most frequently mis-used part-of-speech tags and grammar patterns?

We have shown the effectiveness of the ensemble tagger at the level of both individual words and full identifier names. In this research question, we are interested in understanding the *weaknesses* of our ensemble; where can it be improved in future work and what types of identifiers is it more likely to get wrong? To answer this question, we calculated the patterns that were most frequently mis-annotated by our ensemble along with the frequency of these patterns in our data. We then divided the number of mis-annotations by the pattern frequency to get the proportion and sorted from largest to smallest. Table 10.15 shows the top five mis-annotated grammar patterns per ensemble configuration along with the frequency and proportion information discussed above.

The data in this table shows some consistently mis-annotated patterns. In particular, *NM NM NM+ N*, *PRE NM* N*, and *V NM NM N* are all in the top 5 for each ensemble configuration. Where '+' means "one or more" of the annotation to its left and '*' means "zero or more" of the annotation to its left. A high frequency of mis-annotating *PRE NM* N* is unsurprising due to the fact that most ensemble configurations had trouble with annotating PRE; the best ensemble configuration achieving only .043 F1. Note that this low F1 score means that it **both** mis-annotates some NM+ N patterns by annotating PRE where it should have annotated NM as well as mis-annotating PRE NM+ N patterns as NM+ N; not recognizing the first word as a PRE and instead annotating NM. This fact helps explain one of the other patterns it frequently mis-annotates– the elongated noun-phrase patterns (*NM NM NM+ N*), since it is typically mis-annotating the leftmost NM as PRE. The other pattern it gets wrong frequently is the elongated verb phrase pattern. The ensemble seems to get shorter verb phrase patterns correct but the longer they become the harder it becomes for the ensemble to annotate them. One reason for this is likely the lack of verb phrases that are greater than four and five words in the training set. This may also have something to do with the fact that we use **normalized position** as one of our features, as discussed in Section10.2.6. This feature normalizes the length of an identifier by considering words to one be in one of three places: the beginning, the middle, or the end. This helps recognize the fact that the first and last words in an identifier are more likely to be a specific annotation (e.g., the last noun in an identifier is usually a head-noun, whereas middle-nouns are typically noun-adjuncts).

We manually looked at examples of each of these commonly mis-annotated patterns to understand the characteristics of these types of identifiers that the ensemble finds confusing. To make this analysis simpler, we will focus on the best ensemble configurations: DTCP and RFCP. Our manual analysis of the data shows that the most confusing factor in most of these patterns for both RFCP and DTCP is *PRE*. That is, when these are mis-annotated, it is because the correct annotation contains a preamble. For example, *eglewAndroidFrameBufferTarget* and *mRemoveUserDataResponseArgs* both have a grammar pattern that begins with a Preamble but they are mis-annotated as *NM NM NM NM+ N*. The one exception is the *V NM NM N* pattern, which tended to be mis-annotated because the correct pattern does not follow a standard verb-phrase pattern. These are function names like *clampFixMaxcolor* (*fix* is an abbreviation for *fixpoint*) and *ActionViewShowMasterPages*, which have non-standard function naming structure; *V N NM N* and *NM N V NM NPL* respectively. This does follow results from prior work [177], as we found that functions have the largest number of unique grammar patterns; many function names may follow a non-standard format, and the further they are from a standard verb phrase, the more difficult it may be for our tagger to annotate it correctly. If the reader is interested in common mis-annotations made by POS taggers, please refer to prior work for more information on the types of mistakes they make frequently [177].

Discussion of Feature Importance

The results to this RQ show us that more context, in the form of features, is likely required to increase the accuracy of the ensemble. Specifically, context that can 1) help identify Preambles, such as word frequency or system naming conventions. And 2) identify stereotype-like [127] information that could tell the ensemble when it might see a function name that does not follow verb phrase patterns. Of course, other types of context may also be helpful, but these are two types of context that, based on our observations, would highly-likely increase the accuracy of our ensemble.

In summary, our ensemble has more trouble with longer identifier names; particularly longer verb-phrase identifiers in general and longer noun-phrase identifiers which contain a preamble. We found that the mis-annotated verb phrase identifiers are typically function names that do not follow a standard verb-phrase structure, while the mis-annotated noun phrases tended to be elongated and contain a preamble. By far, the most confounding factor for our ensemble is when an identifier contains a preamble. Improved Preamble detection is possible, and we plan to address it in the near future, but it requires an analysis of the system-to-be annotated before commencing annotations.

Research Focus Area: Developer Workflow Integration

Chapter 11

IDE Identifier Name Appraisal and Recommendation Plugin

11.1 Overview

The findings from my empirical studies show patterns in how developers craft identifier names and what they consider a strong or high-quality name. As my studies show, automatically determining the meaning of terms (or words) in an identifier is challenging. However, the grammar pattern of the name provides a more feasible mechanism to analyze the quality of the name. To this extent, I worked with my advisor in sponsoring and leading a team of undergraduate software engineering students[1] in implementing an IntelliJ IDEA plugin that incorporates heuristics to provide developers with real-time appraisals and recommendations. The plugin utilizes my custom part-of-speech tagger to determine the grammar pattern of the selected identifier and then based on heuristics. These heuristics include valid grammar patterns and the identifier's relationship to the surrounding code (e.g., data type) to determine if the developer needs to alter the name of the identifier. If the name is of poor quality, the tool recommends an alternate grammar pattern, including highlighting terms in the name that should be removed. The plugin also explains the proposed recommendation.

Figure 11.1 shows a screenshot of the plugin in use. If an identifier's name violates a naming rule, the plugin underlines the name using a green squiggly line.Additionally, on clicking/selecting the name, the UI of the plugin updates to show the current grammar pattern of the name and the recommended grammar pattern, if any. Terms in the name that should be removed appear in red font, while part-of-speech tags that should be added to the name appear in green font. Furthermore,

[1]`https://lll.maxkipust.com/`

the plugin also provides an example and an explanation for the proposed recommendation. Finally, from Figure 11.2, it can be seen that the plugin also provides developers with a summary of all naming violations in the source file.

11.1.1 Technical Design

Figure 11.3 provides a high-level view of the architecture of the name appraisal and recommendation plugin. The plugin comprises of four main parts– (1) Grammar Pattern Appraisal, (2) Grammar Pattern Recommendation, (3) Examples & Explanations, and (4) Identifier Exclusion. The plugin utilizes the ensemble tagger to generate the part-of-speech tags for each term in the identifier's name. The ensemble tagger is made available as a python-based web service. Furthermore, to promote portability, the web service and ensemble tagger are contained within a Docker container.

11.1.2 Grammar Patterns & Rules

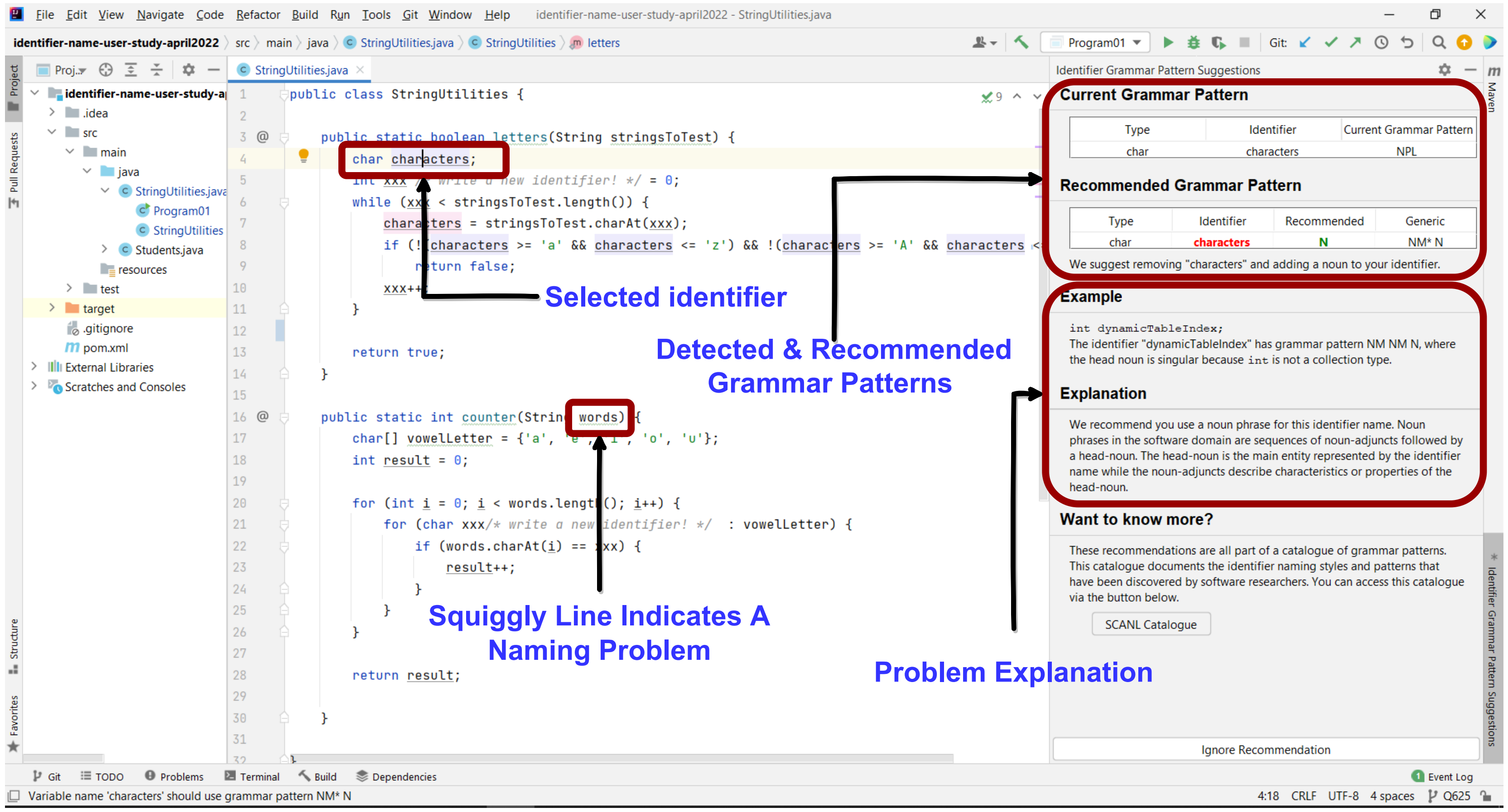
Findings from my empirical studies have shown the existence of a set of common grammar patterns that developers utilize when crafting an identifier's name. Furthermore, these patterns are also associated with specific types of identifiers and contexts. These heuristics are incorporated into the plugin to perform appraisals and recommendations on an identifier's name structure. Table 11.1 discusses common identifier naming patterns and their meaning.

11.2 Plugin Evaluation

To understand the effectiveness and usability of the tool in a real-world setting, I conducted a user study with undergraduate and graduate students. The participants were Golisano College of Computing and Information Sciences students at Rochester Institute of Technology. In total, I recruited 20 participants. I divided the participants into two groups of equal size– Group A and Group B. Participants in each group were given the same Java code snippets, which they had to review in IntelliJ IDEA and make necessary corrections to the identifier names. Participants in Group A had access to the plugin, while participants in Group B (our control group) did not have access to the plugin. Participants in both groups had to complete a pre- and post-questionnaire (the templates are attached in Appendix B).

11.2.1 Methodology

The workflow of the activity consisted of four main parts:


File Edit View Navigate Code Refactor Build Run Tools Git Window Help
identifier-name-user-study-april2022 - StringUtilities.java
identifier-name-user-study-april2022 › src › main › java › StringUtilities.java › StringUtilities › letters
Program01
Git:
Project
Pull Requests
Structure
Favorites
identifier-name-user-study-a
.idea
src
main
java
StringUtilities.java
Program01
StringUtilities
Students.java
resources
test
target
.gitignore
pom.xml
External Libraries
Scratches and Consoles
StringUtilities.java
Selected identifier
Detected & Recommended Grammar Patterns
Squiggly Line Indicates A Naming Problem
Problem Explanation
Identifier Grammar Pattern Suggestions
Maven
SCANL Catalogue
Ignore Recommendation
Git
TODO
Problems
Terminal
Build
Dependencies
Event Log
4:18 CRLF UTF-8 4 spaces Q625

```
public class StringUtilities {

    public static boolean letters(String stringsToTest) {
        char characters;
        int xxx /* write a new identifier! */ = 0;
        while (xxx < stringsToTest.length()) {
            characters = stringsToTest.charAt(xxx);
            if (!(characters >= 'a' && characters <= 'z') && !(characters >= 'A' && characters <
                return false;
            xxx++;
        }

        return true;
    }

    public static int counter(String words) {
        char[] vowelLetter = {'a', 'e', 'i', 'o', 'u'};
        int result = 0;

        for (int i = 0; i < words.length(); i++) {
            for (char xxx/* write a new identifier! */  : vowelLetter) {
                if (words.charAt(i) == xxx) {
                    result++;
                }
            }
        }

        return result;

    }
```

Current Grammar Pattern

Type	Identifier	Current Grammar Pattern
char	characters	NPL

Recommended Grammar Pattern

Type	Identifier	Recommended	Generic
char	characters	N	NM* N

We suggest removing "characters" and adding a noun to your identifier.

Example

`int dynamicTableIndex;`
The identifier "dynamicTableIndex" has grammar pattern NM NM N, where the head noun is singular because `int` is not a collection type.

Explanation

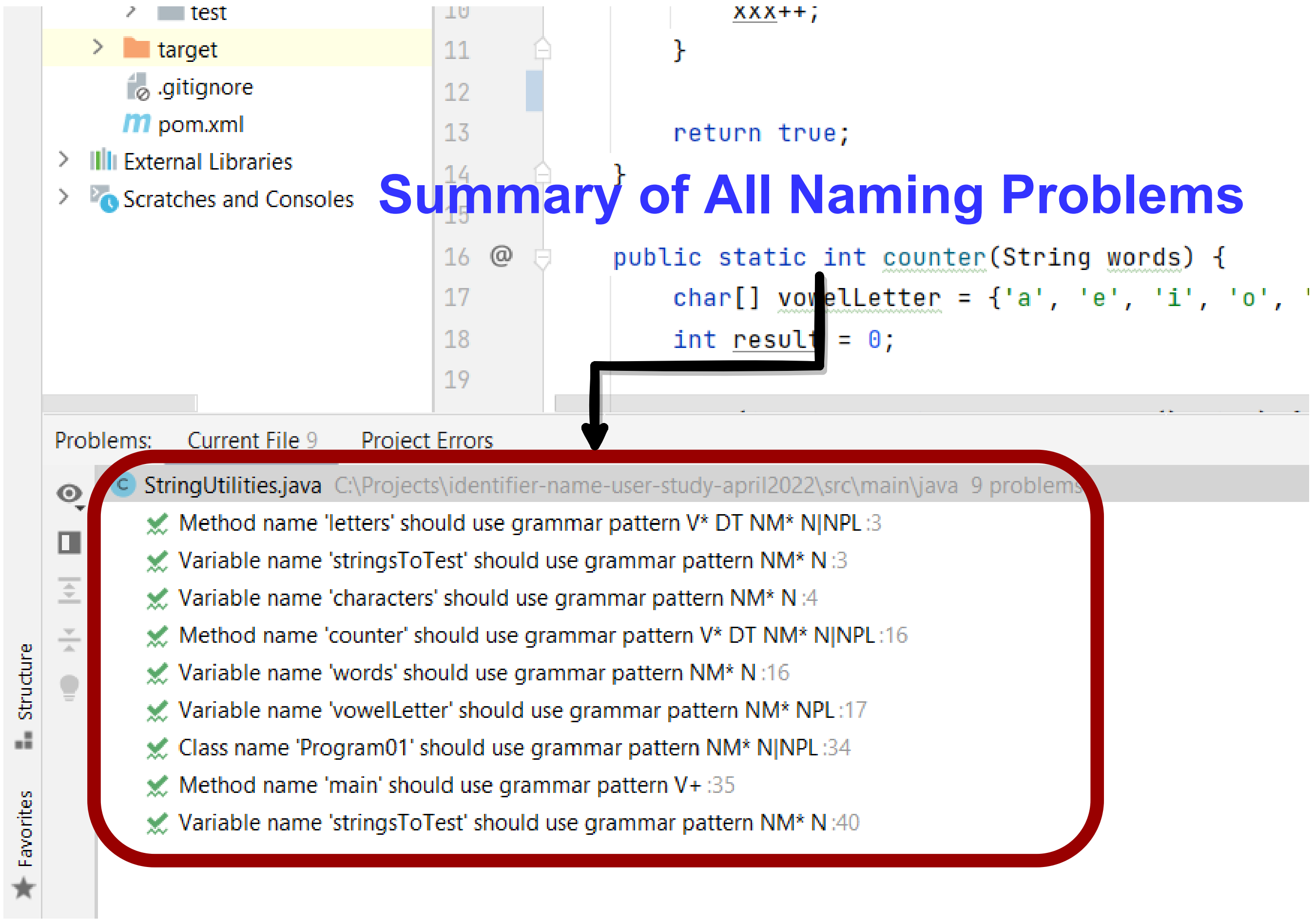
We recommend you use a noun phrase for this identifier name. Noun phrases in the software domain are sequences of noun-adjuncts followed by a head-noun. The head-noun is the main entity represented by the identifier name while the noun-adjuncts describe characteristics or properties of the head-noun.

Want to know more?

These recommendations are all part of a catalogue of grammar patterns. This catalogue documents the identifier naming styles and patterns that have been discovered by software researchers. You can access this catalogue via the button below.

Variable name 'characters' should use grammar pattern NM* N

Figure 11.1: The user interface of the IDE name appraisal and recommendation plugin.



```
        xxx++;
    }

    return true;
}

public static int counter(String words) {
    char[] vowelLetter = {'a', 'e', 'i', 'o', '
    int result = 0;
```

StringUtilities.java C:\Projects\identifier-name-user-study-april2022\src\main\java 9 problems

```
Method name 'letters' should use grammar pattern V* DT NM* N|NPL :3
Variable name 'stringsToTest' should use grammar pattern NM* N :3
Variable name 'characters' should use grammar pattern NM* N :4
Method name 'counter' should use grammar pattern V* DT NM* N|NPL :16
Variable name 'words' should use grammar pattern NM* N :16
Variable name 'vowelLetter' should use grammar pattern NM* NPL :17
Class name 'Program01' should use grammar pattern NM* N|NPL :34
Method name 'main' should use grammar pattern V+ :35
Variable name 'stringsToTest' should use grammar pattern NM* N :40
```

Figure 11.2: Summary of all naming problems in the currently open file.

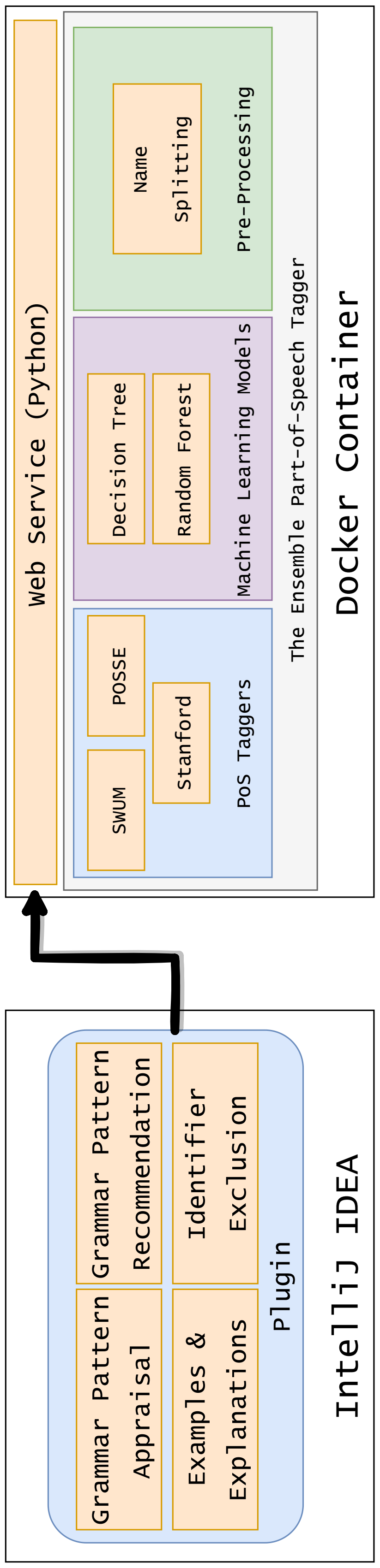


Figure 11.3: High-Level architectural view of the IDE name appraisal and recommendation plugin.

Table 11.1: Common grammar patterns utilized in structuring identifier names.

Grammar Pattern Sequences	Definition
NM* N	**Noun Phrase**: Zero or more noun-modifiers appear to the left of a head-noun. Noun-modifiers that appear before the head-noun act as a way to specialize our understanding of the head-noun by taking the general concept the head-noun represents and reducing it to a more concise, specific concept. For example, in the identifier 'issueDescription' the head-noun is 'Description', which is the general concept. The noun-adjunct, 'issue', specializes our understanding of the 'Description' by specifying what kind of 'Description' we are talking about. It is good practice to be careful in the choice, and number, of noun-modifiers to use before the head-noun. A good identifier will include only enough noun-modifiers to concisely define the concept represented by the head-noun. This is the most common naming pattern for identifiers that are not function names.
NM* NPL	**Plural Noun Phrase**: This is identical to Noun Phrase (NM* N), except the head-noun is plural. The plural is often purposeful in that the head-noun's plurality expresses the multiplicity of the data. That is, these identifiers (when they are not function names) are more likely to have a collection data type. Some naming conventions (e.g., the Java naming standard) generally consider it good practice to match the plurality of the identifier with whether its type represents a singular or collection object. Identifiers that follow this pattern are usually not function names.
V NM* (N—NPL)	**Verb Phrase**: The addition of a verb to a noun phrase creates a verb phrase. The verb in a verb phrase is an action being applied to (or with) the concept embodied by the noun phrase that follows. In some cases, instead of being an action, the verb is an existential quantifier. In this case, the identifier's data type is probably (interpretable as) Boolean. These are typically either function identifiers or identifiers with a boolean type.
NM* N P NM* (N—NPL)	**Prepositional Phrase With Leading Noun Phrase**: Sometimes a noun phrase is explicitly present on both the left and right of the preposition. When the left-hand-side noun-phrase is specified, there is an explicit relationship between the left- and right-hand side noun-phrases. This relationship is expressed through the preposition. The preposition helps us understand how the entity (or entities) represented by both noun-phrases are related in terms of order, space, time (e.g., generated_token_on_creation), ownership (e.g., scroll_id_for_node), causality, or representation (e.g., url_from_json, query_timeout_in_milliseconds). This pattern is used in many types of identifiers whether they are function names or otherwise.
V P NM* (N—NPL)	**Prepositional Phrase With A Leading Verb**: Same as prepositional phrase pattern but the leading verb, or verb phrase, is specified this time. As before, the preposition helps us understand how the entity (or entities) represented by the verb- and noun-phrases are related in terms of order, space, time, ownership, causality (e.g., destroy_with_parent), or representation (e.g., save_as_quadratic_png, tessellate_to_mesh, convert_to_php_namespace). The usage of this pattern is similar to when the verb is implicit. There may still be an implicit noun phrase to the right of the verb and to the left of the preposition. This pattern is used in many types of identifiers whether they are function names or otherwise.
V* DT NM* (N—NPL)	**Noun Phrase With A Leading Determiner**: The addition of a determiner tells us how much of the population, which is specified by the noun-phrase, is represented, or acted on, by the identifier. Typically, the determiner will tell us that we are interested in ALL, ANY, ONE, A, THE, SEVERAL, etc., of the population of objects specified by the noun phrase. If there is a leading verb, the verb specifies an action to take on the population or it represents existential quantification (e.g., matchesAnyParentCategories). This pattern is used in many types of identifiers whether they are function names or otherwise.
V+	**Verb Sequence**: One or more verbs with no noun phrase. Because these are missing a noun phrase to act upon (in contrast to the Verb Phrase pattern above), a larger population of these are likely generic functions like Sort (though more data/research is needed), which can act upon many different types of data and have different behaviors depending on the data being acted upon. The noun phrase that this action (i.e., the verb) is applied to is implicit. That is, it is not present in the identifier name. Instead, the noun phrase is implied by the program context (e.g., it is represented by a this-pointer) or it is present in the function parameters. In some cases, these are boolean-type variables that may be missing an existential quantifier (e.g., add 'is' before 'parsing' to make it explicit). These are typically function names or identifiers with a boolean type.

1. Each participant was given a 5-minute explanation about part-of-speech tags.
2. The participant had to complete a pre-questionnaire.
3. The participant then conducted the identifier naming evaluation activity using IntelliJ IDEA.
4. Finally, the participant completed a post-questionnaire. While completing the post-questionnaire, the participant could refer to the code snippets but could not make any changes to them.

Details of each stage are described below.

Pre-Questionnaire.

This questionnaire captured the skill and experience level of the participant. Group A participants were also asked about the usefulness of a real-time grammar pattern-based naming suggestion tool.

Identifier Name Evaluation Activity.

In this activity, the participants utilized IntelliJ IDEA to comprehend two working programs. As part of the activity, the participants updated the identifier names in instances where they felt the name did not reflect its intended behavior. One program contained methods for string manipulation, while the other program was a simple object-oriented program. In both instances, there exist identifiers with poor or low-quality names. The participants were permitted to run the code if it helped them better understand the program's functionality. However, the participants were not allowed to ask the investigator to explain the workings of the programs.

Post-Questionnaire.

This questionnaire solicited feedback regarding the naming activity. Both groups were asked questions about certain identifier names in the code and general questions about the importance of part-of-speech tags. Additionally, Group A participants were asked about the usefulness and effectiveness of the plugin as well as areas for improvement.

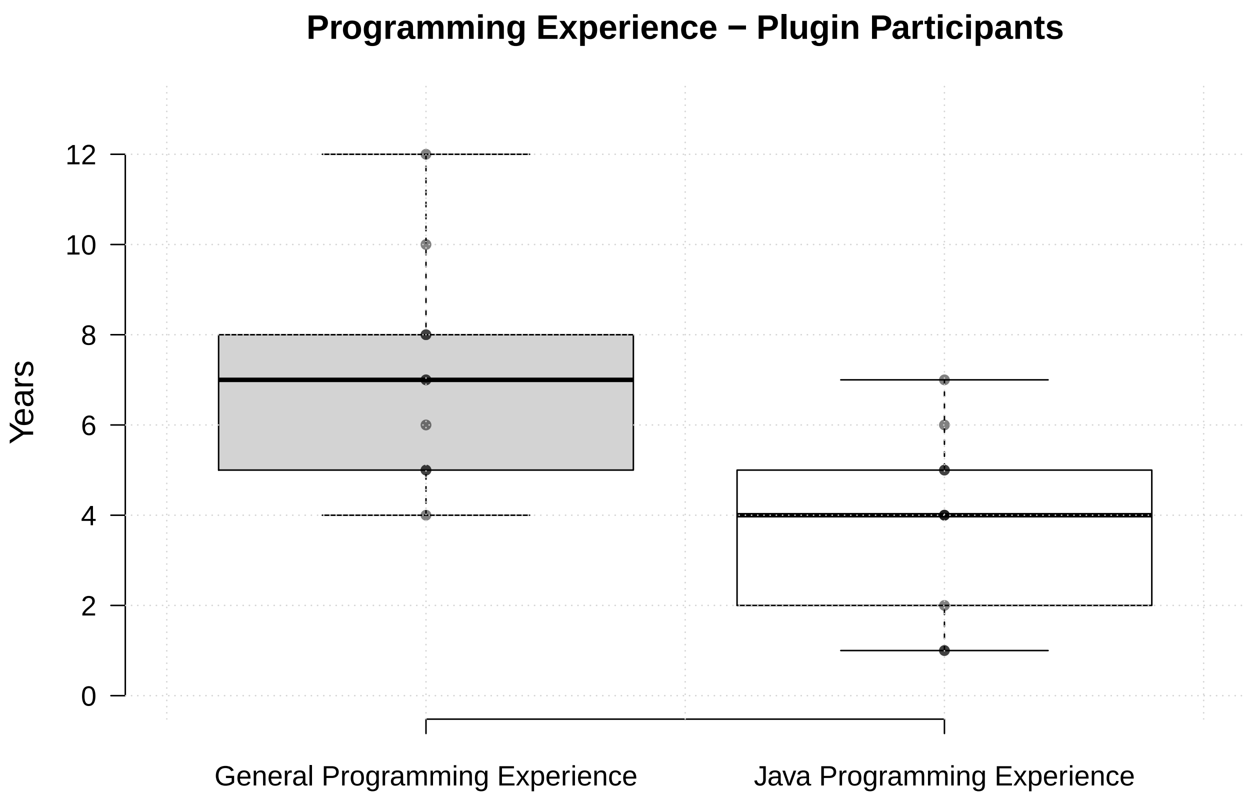


Figure 11.4: Boxplot showing the programming experience of participants that utilized the IDE plugin.

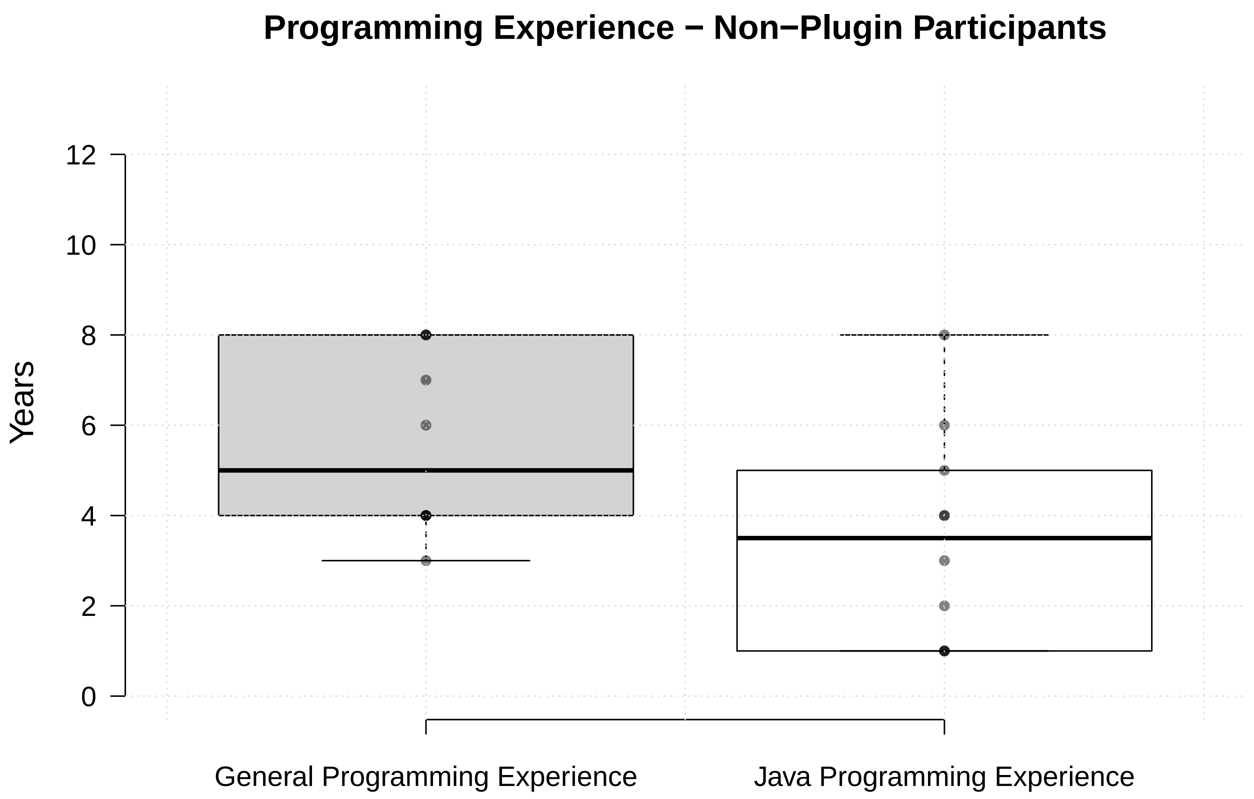


Figure 11.5: Boxplot showing the programming experience of participants that did not utilize the IDE plugin.

Table 11.2: Participant's hands-on familiarity with IntelliJ IDEA

IntelliJ IDEA Hands-on Familiarity				
Extremely	**Very**	**Moderately**	**Slightly**	**Not at all**
		Plugin Participants		
1	3	4	1	1
		Non-Plugin Participants		
0	3	6	1	0

11.2.2 Results

Skill and Experience

Figure 11.4 and 11.5 show a boxplot distribution of the programming experience of participants that did and did not utilize the IDE plugin, respectively. On average, participants in Group A had 7.2 years of general programming experience and 3.9 years of Java programming experience. Participants in Group B had an average of 5.6 and 3.5 years of general and Java programming experience, respectively. Regarding familiarity with IntelliJ IDEA, as shown in Table 11.2, most participants in both groups rated their familiarity from very to moderately.

Quantitative Feedback

In the post-questionnaire, the participants were asked general questions about crafting identifier names. The majority of participants in both groups seriously consider the part-of-speech tags when crafting identifier names. To be more precise, the priority ranges from medium to high priority, as shown in Table 11.3. Looking at the responses provided, it is clear that the participants consider part-of-speech tags an essential part in comprehension activities as it helps in convening the meaning of the identifier. Specifically, approximately 80% of the participants rated their priority on part-of-speech tags as either High Priority or Essential.

In addition to the general questions, Group A participants were asked questions specific to the plugin. From Table 11.4, 80% of the participants rated the convenience of having a grammar pattern recommendation tool as either Convenient or Very Convenient. Furthermore, as shown in Table 11.5,70% of the Group A participants rated their ability to interpret the recommendations as either Easy or Very Easy. Finally, an overwhelming majority (i.e.,90%) of the plugin users rated the accuracy of the plugin's recommendations as either Satisfied or Very Satisfied (refer to Table

Table 11.3: Priority a participant places on part-of-speech tags when crafting an intensifier's name.

Priority on part-of-speech tags of a name				
Not a priority	**Low priority**	**Medium priority**	**High priority**	**Essential**
		Plugin Participants		
0	1	3	5	1
		Non-Plugin Participants		
0	0	0	6	4

Table 11.4: Participants who utilized the plugin rate the convenience of a real-time grammar pattern-based name suggestion tool before and after the naming activity.

Convenience of a real-time grammar pattern-based name suggestion tool				
Very convenient	**Convenient**	**Neutral**	**Inconvenient**	**Very inconvenient**
		Before Naming Activity		
2	6	2	0	0
		After Naming Activity		
1	7	2	0	0

11.6).

Qualitative Feedback

In addition to capturing Likert-based quantitative data, the post-questionnaire also captured qualitative feedback from the participants. A thematic analysis was performed on the user feedback, which was then grouped into one of three categories – (1) Positive Feedback, (2) Negative Feedback, and (3) Enhancements.

Positive Feedback: While the participants found that the recommendations were satisfactory, they also felt that the tool would be beneficial for novice developers to ensure adherence to consis-

Table 11.5: The rating a participant provides on the ease of interpreting the recommendations proposed by the plugin.

Ease of interpreting the recommendations proposed by the plugin				
Very difficult	**Difficult**	**Neutral**	**Easy**	**Very easy**
0	2	1	6	1

Table 11.6: The satisfaction the participant places on the accuracy of the plugin's recommendations.

Satisfaction with the accuracy of the recommendations				
Very dissatisfied	**Dissatisfied**	**Unsure**	**Satisfied**	**Very satisfied**
0	0	1	7	2

tency in identifier names. Furthermore, the participants also feel that the plugin forces the user to think about the quality of the identifier's name, and the examples and explanations help understand the problems with names.

Negative Feedback: As this plugin is still in the prototype stage, some participants encountered noticeable performance issues with the IDE behaving sluggishly and hanging at times. As the plugin's purpose is to appraise and recommend grammar patterns, the developer would require an understanding of part-of-speech tags. However, this is not always the case and requires additional effort for developers to learn or ramp-up on part-of-speech concepts. Finally, not all grammar pattern recommendations were accurate.

Enhancements: While the participants appreciated the examples and explanations provided by the plugin, most of the participants suggested that the plugin should provide more examples to help understand the naming problem. Participants also suggested that the UI include definitions of part-of-speech tags as not everyone is familiar with the concepts. Lastly, the participants also requested improvements to the UI to make it easier to navigate to identifiers in the code.

Chapter 12

Discussion

Through multiple studies, I have constructed a novel approach to assist developers with real-time identifier name appraisal and recommendation. Through empirical studies, I have accumulated and analyzed data on how developers craft and evolve the name of an identifier. Using the findings from these studies, I have constructed tools that can be utilized by the developer and research communities to aid in the research and maintenance of identifier names. In this section, I summarize the research I have conducted for my dissertation, the contributions these studies make to the field, and how they help answer my proposed research questions.

12.1 Overall Findings

12.1.1 Identifier Name Evolution

Rename Prevalence

A prelude to studying identifier names is first to understand the extent to which developers perform rename operations on identifiers (i.e., rename refactorings). To this end, I have analyzed an extensive set of open-source Java systems. These systems include Android applications and well-engineered non-mobile systems.

Key Findings: My studies confirm prior work showing that rename refactorings are highly prevalent in mobile applications [184, 195] and traditional systems [91, 191, 192]; with developers frequently applying renames to attributes. These findings show that the simple task of performing a rename is also one of the most frequent areas of rework faced by developers– in most cases, rename refactorings contribute to over 40% of the refactorings applied by developers.

Contribution: Renaming is the most common refactoring operation applied by developers and available in IDEs, and yet there is very little support to help developers use high-quality wording and identifier structure. These studies, and others, function as motivation for my work; they show the ubiquity of renaming operations. My publicly available datasets contain refactoring operations applied to hundreds of Android apps and well-engineered traditional Java systems.

Semantic Evolution

Once I determined that renames are indeed prevalent in source code, I examined the semantic change a name undergoes during a rename [190, 191, 192]. Studying the semantics of a name is essential as it shows how the name changes and thereby assists me in understanding mechanisms/factors I need to consider when evaluating term replacements or suggesting term replacements for an identifier name. In these studies, I examine each rename refactoring and categorize them into the different types prescribed by rename-specific taxonomy. The taxonomy includes three high-level categories: rename form, semantic change, and grammar change.

Key Findings: In two of my studies [191, 192], I show that developers frequently narrow the meaning of identifier names when renaming. I also show that most of the renames are simple (i.e., only a single term changes in the rename operation). In these two studies, I looked at identifiers irrespective if they were production code or part of the test suite. However, in a study focussing on methods of test cases [190], I observed that developers prefer to perform a change in the meaning of the identifier's name. I also encounter challenges utilizing standard natural language processing techniques to analyze terms in an identifier's name. These challenges include the presence of domain terms, misspellings, and preambles, among others.

Contribution: In addition to multiple datasets containing the semantic analysis of a large volume of identifier names, my findings act as the first step toward better supporting tools for identifier name appraisal and evolution.

Contextualizing Renames

To understand the decision-making process behind applying a rename, I contextualized the semantic change made to an identifier's name using the semantic meaning of an identifier's name, commit log data, co-occurring refactorings, and the associated data type [185, 191, 192]. I used the commit log to understand why a rename was applied, while co-occurring refactorings and data type changes helped me understand how well-defined code changes affect name evolution.

Key Findings: Developers generally narrow the name of the identifier in conjunction with a change in data type. Furthermore, when a data type is modified such that it becomes a collection, there is a change in plurality for its corresponding identifier name. Furthermore, in most scenarios, renaming of an element does not generally seem to be influenced by, nor does itself influence another type of refactoring on the same element. However, for the subset of refactorings, a rename occurs in the commit directly following a Move Attribute. In my studies, I also highlighted challenges encountered analyzing commit logs to derive the motivations behind rename operations. For instance, while the commit message does indicate a modification occurred to the source code, the message does not indicate the rationale for the change. Additionally, the application of standard topic modeling techniques to commit logs does not yield adequate topics.

Contribution: These studies identify how the context surrounding a rename affect the rename itself. In particular, this work shows that data type changes influence the plurality of identifier names, that renames typically specialize in the meaning of an identifier instead of generalizing or removing meaning, and that renames are more highly likely to follow certain types of refactorings versus others.

Identifier Grammar Patterns

In studies on grammar patterns, I investigated the part-of-speech tags in identifier names for test [190] and production files [177]. These studies involved the manual annotation of part-of-speech tags in identifier names. The study of test suites involved looking at how the grammar patterns of test methods evolve (i.e., comparing the part-of-speech tags before and after the rename). I also compared the detected grammar patterns, in test suites, against other known patterns and production patterns. The study also looked at the frequent terms that are replaced in a rename.

Key Findings: There is a difference in structure between test and production method names. Further, test method name refactorings tend to change the meaning of terms in the name. I also identified additional grammar patterns developers utilize to craft test method names and relationships between specific part-of-speech tags and source code statements. This work also identified common words and phrases which are synonymous in test method renames.

Contribution: Two datasets are made publicly available. The first is a gold set of 1,335 manually-annotated (and validated) grammar patterns for five identifier categories: class, function, declaration-statement, parameter, and attribute names. The second dataset is a manual annotation of part-of-speech tags for 615 test method names. It augments our understanding of common and diverse grammar patterns found in identifier names. My work leverages these patterns to improve part-of-speech tagging and assist in renaming by, in part, recognizing when names deviate from these patterns.

12.1.2 Tool Development

Rename Semantics Detection Tool

To facilitate my research on semantic changes identifier names undergo, I constructed a tool that automates the detection of the semantic and form changes a name undergoes when renamed. I utilize this tool in several of my studies [185, 190, 191, 192] to understand how the name of an identifier evolves throughout the lifetime of the system. The tool determines the semantic relationship (e.g., synonyms, hypernyms, etc.) between the old and the new name of the identifier using natural language-based techniques.

Contribution: While this tool is primarily suited for studying renames after they have been applied, some aspects of this tool are utilized in my name appraisal and recommendation plugin. Specifically, this contribution is useful for producing historical data about how names evolve, which is important for understanding how renames have been applied historically. This provides me with data I can use to make data-driven wording and structure recommendations during a rename.

Linguistic Anti-Pattern Detection Tool

My research has also led to the construction of a tool that utilizes static analysis to detect linguistic anti-patterns in the source code of Java and C# systems [189]. The tool outperforms an existing tool and detects more linguistic anti-patterns than the existing tool. In addition to detecting these anti-patterns, this command-line tool also provides an explanation of the problem to help developers understand the mistakes that they are making when naming identifiers.

Contribution: An open-source identifier naming violation appraisal tool that developers can integrate into their workflow, such as the build process, to detect the presence of linguistic anti-patterns in their code and explanations of the problem.

Ensemble Part-of-Speech Tagger

As highlighted in my research on identifier evolution, standard natural language processing techniques are not adequate to analyze source code. To this extent, I have contributed to the construction of an ensemble part-of-speech tagger that analyzes identifier names. The tagger uses machine learning and the output from multiple part-of-speech taggers to annotate natural language text. The ensemble uses three state-of-the-art part-of-speech taggers: SWUM, POSSE, and Stanford. Finally, the ensemble achieves 75% accuracy at the identifier level and 84-86% accuracy at the word level.

Contribution: A part-of-speech tagger specifically oriented for source code, than any other approach currently available, will improve research involving linguistic-based analysis of source code, such as identifier name appraisal and recommendation. This ensemble tagger can be integrated into other tools to support identifier naming research and maintenance.

12.1.3 Developer Workflow Integration

IntelliJ IDE Identifier Name Appraisal and Recommendation Plugin

Based on my prior empirical and tool studies, I have worked on constructing an IDE plugin that provides developers with real-time appraisals and recommendations of identifier name structures (Chapter 11). The plugin utilizes the ensemble part-of-speech tagger to determine the grammar pattern associated with the identifier's name and, based on heuristics, determines if the name needs improvement. The plugin recommends grammar patterns along with examples and an explanation for the recommendation.

Key Findings: A user study showing that the recommendations provided by the plugin are mostly acceptable to users and the examples and explanations helped in understanding the problem with the original name and how it could be improved. Furthermore, the plugin is useful for novice developers as it will help them to ensure consistency in identifier naming in the source code.

12.2 Research Question Analysis

12.2.1 RQ$_1$: How effectively, in terms of correctness, can we generate identifier name structure recommendations?

Along with prior work in this area, my studies show that while there is a possibility to determine the quality of an identifier's name by evaluating the words that make up the name, there are challenges with automatically determining the meaning of these words and how they interact with one another. Words are diverse and subjective; for example, a single word can have multiple meanings and would require a human to determine the correct meaning based on usage (i.e., context). In English prose, the context is specified in natural language, but this is not always the case in code, where the context may be within the code's behavior. For instance, the identifier named "do forward" can be used as a synonym for "redirect" (as in redirecting an HTTP request) or can also be used to mean "advance" (as in moving a figure on the screen). The exact meaning of the name can only be known by analyzing the code, such as by examining the API in use.

Part-of-speech tags, on the other hand, are more constrained, and so is determining the words that are part of each category. Furthermore, similar to English prose, there is consistency in how developers structure names with regards to their semantics. For instance, methods are usually structured using the verb phrase, as in the case of the method name "setEmployeeName", which has a *V NM N* structure. Now, coupled with static analysis techniques, it becomes feasible to determine if the grammar pattern is utilized in the proper context. For instance, collection types are associated with a plural noun phrase, such as with "String[] tv_channel_frequencies" that has the *NM NM NPL* grammar pattern.

In my studies, I have shown that existing state-of-the-art, off-the-shelf part-of-speech taggers are unsuitable for source code. The correctness of these taggers performs poorly on identifier names. To this extent, I have worked on implementing a specialized ensemble part-of-speech tagger for identifier names. With an accuracy of 75% at the identifier level and 86% accuracy at the word level, the ensemble tagger outperforms the state-of-the-art taggers. That said, further improvements can be made to the tagger to improve its effectiveness in generating accurate tags.

Through my user study, I have shown that the heuristics incorporated into the IDE plugin does in fact align with the concepts developers associate with high-quality names, such as names associated with a collection data type having a plural noun phrase grammar pattern.

In summary, this novel approach of exploiting the relationship between an identifier's grammar pattern and the surrounding code makes it possible to provide accurate name appraisals and recommendations without relying on the meaning of the words in an identifier's name.

12.2.2 RQ$_2$: To what extent does an automated mechanism, based on the semantic structure of a name, positively or negatively influence naming practices?

To determine the usability of my proposed approach, I have worked on constructing an IntelliJ plugin that provides developers with real-time appraisals and recommendations on the semantic structure of identifier names. The plugin employs a static analysis approach that utilizes the specialized ensemble tagger and heuristics, defined in my empirical studies, to assist developers with crafting and maintaining identifier names in the source code. To understand the effectiveness of the plugin from correctness to usability, graduate and undergraduate students in computing were recruited to participate in a user study.

In general, of the 20 participants enrolled in the study, 80% of them believe it is either a high priority or essential for the consideration of part-of-speech tags when composing an identifier's name. This finding shows that my novel approach of using grammar patterns to assist developers with maintaining high-quality identifier names is indeed a viable approach and deserves further research.

The user study shows that the participants welcome the plugin as it ensures consistency in identifier naming within the project and feel it would immensely help novice developers. This ties in with findings from my empirical studies that show novice developers tend to perform more rename refactorings than other refactoring operations. Furthermore, 80% of the participants agreed that having a grammar pattern recommendation tool in their toolbox is convenient when coding and were satisfied with the plugin's recommendations. However, at the same time, they also observe instances for tagger improvements, such as the method "stringToTest()", where "Test" is tagged as a noun instead of a verb. Finally, while most participants appreciated the examples and explanations, they suggested that the plugin should show more examples as not all developers are knowledgeable about part-of-speech tags.

In summary, by incorporating my empirical findings into a real-time name appraisal and recommendation plugin, I have established a mechanism to integrate my work into the developer workflow. Without having to leave the IDE or switch between tools, the developer can craft and maintain high-quality identifier names in their project to a high degree of accuracy.

12.2.3 RQ$_3$: What are the primary challenges in appraising and recommending the semantic structure of identifier names, and how can these be improved?

As stated in my empirical studies, identifier names are both, diverse and subjective. Furthermore, with diverse sets of technologies, domains, developers, and projects, it becomes challenging to arrive at a one-stop solution for recommending and appraising high-quality identifier names. To further complicate matters, standard natural language processing techniques are not specialized to handle software engineering artifacts, including identifier names in the source code.

To this extent, the tools I have constructed, specifically the ensemble part-of-speech-tagger and IDE plugin, while improving the developer code comprehension experience, do not provide a complete solution to the problem of identifier naming.

With an F1 score of .043, preamble detection by the ensemble tagger is one area for improvement. This tag is difficult for the trained taggers within the ensemble to recognize, thereby impacting the ensemble. The taggers annotate the first word as a noun modifier instead of a preamble; this, in turn, results in the following consistently misannotated patterns: *PRE NM* N* and *NM NM NM+ N*. One possible approach to address this issue is to analyze the system to be annotated before commencing annotations. Specifically, to detect preambles, we have to detect frequently-occurring identifier prefixes and then determine whether those prefixes follow the preamble rules.

The other pattern it gets wrong frequently is the elongated verb phrase pattern (i.e., *V NM NM N*). The ensemble gets shorter verb phrase patterns correct, but the longer they become, the harder it becomes for the ensemble to annotate them. One reason for this is likely due to the lack of verb phrases greater than four and five words in the training set.

Abbreviations and acronyms are also a pain point for the tagger. For instance, in the identifier name "clampFixMaxcolor", the word Fix is tagged as a verb; this word is an abbreviation for "fixpoint" and should be a noun modifier. The system will need to be analyzed in advance to expand abbreviations and acronyms in the code to handle this challenge.

When conducting our user study, one observation we encountered was the tag generated for the word "test". For instance, in the name of the parameter "stringsToTest", the tagger annotates this word as a noun instead of a verb. However the method "testEmployeeName()" is correctly tagged. Additionally, the plugin treats a constructor as a method and incorrectly informs the developer that the name should begin with a verb when it should not. Finally, transformation methods that start with a preposition are marked as an issue. For example, the plugin incorrectly recommends the method "toSummaryString" to start with a verb.

Furthermore, when conducting my empirical studies, I have encountered challenges with analyzing identifier names. These challenges range from either lack of tools or shortcomings in existing tools to analyze the semantics of identifier names. For instance, current NLP tools and technologies such as WordNet and the Standford Part-of-Speech Tagger have been built for and evaluated against general English prose. While such technologies can be used for analyzing software engineering artifacts, their accuracy is low. Hence, there is a need for specialized technologies, which I have contributed to with the ensemble tagger; however, more work is needed in this area. The dearth of developer-oriented studies also impacts research into identifier names. Since developers are diverse in terms of experience and skills, their concept of a high-quality name can vary. Hence, recommendation tools need to account for the type of developer when making the recommendation. However, studies that correlate developer experience with identifier name quality are lacking.

12.2.4 Novelty

To conclude this Chapter, my work shows the novel contributions I make to the field in my dissertation. From RQ_1, I show that using grammar patterns, it is possible, at a conceptual level, that when identifiers follow a well-established naming pattern, they reflect both the linguistic and program behavior. Through RQ_2, I show the possibility of incorporating my grammar pattern and heuristic analysis into an IDE plugin to automate the appraisals and recommendations. I show that developers find this seamless integration into the developer workflow both valuable and useable. Finally, through my numerous empirical studies, I have encountered multiple types of challenges involved with analyzing and recommending identifier names. The novelty of RQ_3 are these conceptual challenges that range from lack/shortcomings in quality tools to the shortage of existing research to understand the internal comprehension that developers perform when trying to understand the behavior of the code they are reading.

Chapter 13

Conclusion

Being the atoms of program comprehension, it is essential that developers craft high-quality identifier names to ensure cost-effective maintenance of software systems. To this extent, the research community has proposed multiple approaches to assist developers with crafting and maintaining high-quality names. However, these approaches focus on the semantics or the styling of the name. The work in my dissertation takes a novel approach to the problem of identifier name quality by focusing on the semantic structure of the identifier's name and its relation to the surrounding code.

Through empirical studies, I have outlined a series of grammar patterns developers frequently utilize and heuristics to determine if the patterns are valid based on code behavior. Furthermore, I have also worked on constructing tools, such as a naming violation detector and an ensemble part-of-speech tagger that specializes in annotating identifier names.

To achieve my goal of improving the developer code comprehension experience, I have worked on constructing an IDE plugin that seamlessly integrates into the developer workflow to assist with identifier name maintenance in the project. This IntelliJ plugin incorporates the findings from my empirical studies and the ensemble part-of-speech tagger to provide developers with real-time appraisals and recommendations about the semantic structure of identifiers in the code. Furthermore, to help developers understand their mistakes, the plugin provides examples and explanations about the recommendations. A user study shows the plugin's effectiveness in structuring names with correct grammar patterns, but also highlights room for improvement.

13.1 Future Work

The work I have performed in my dissertation is moving me in the right direction towards achieving my goal of providing developers with the best program comprehension experience possible.

While my completed studies have highlighted heuristics for identifier name appraisals and recommendations, much still needs to be known about the relationship between a name and its surrounding code. I plan on continuing my empirical studies by studying different types of systems and code, such as auto-generated code, to determine how, for instance, a specific domain influences the meaning and structure of a name.

Even though my tools assist developers with maintaining identifier names in the code, they are not a complete one-stop solution to the problem of identifier naming quality. I will continue to work on enhancing the shortcomings of these tools based on user studies with developers of varying experiences, to improve their accuracy and usability.

Finally, a crucial part of my future work is combining static analysis and artificial intelligence (AI) to detect low-quality identifier names. While AI understands trends in the data, it finds it a challenge to determine the context of data usage. My empirical research, on the other hand, provides insight into how terminology changes based on context. Hence, by synergizing my human-curated data with AI, I will be in a position to improve the accuracy of my name evaluation and recommendation plugin and, at the same time, provide developers with a detailed explanation about naming violations.

Bibliography

[1] https://github.com/unquietcode/flapi/commit/4586325.

[2] https://github.com/square/retrofit.

[3] https://github.com/jenkinsci/jenkins.

[4] https://github.com/shadowsocks/shadowsocks-windows.

[5] https://github.com/PowerShell/PowerShell.

[6] /abuchen/portfolio/ui/dialogs/transactions/abstractsecuritytransactionmodel.java. https://github.com/buchen/portfolio/commit/e1d7472.

[7] abuchen/portfolio/ui/dialogs/transactions/buysellmodel.java. https://github.com/buchen/portfolio/commit/9fc2fad.

[8] apache/commons/compress/archivers/zip/generalpurposebittest.java. https://github.com/apache/commons-compress/commit/fa2e5bd.

[9] api/src/main/java/org/openmrs/personaddress.java. https://github.com/openmrs/openmrs-core/commit/fd5ed0d.

[10] apvs/src/main/java/ch/cern/atlas/apvs/client/ui/abstractmeasurementview.java. https://github.com/cern/apvs/commit/c1e5792.

[11] apvs/src/main/java/ch/cern/atlas/apvs/client/ui/measurementview.java. https://github.com/cern/apvs/commit/71fc572.

[12] atlas/api/validation/validators/networkitemvalidatortest.java. https://github.com/openstack-atlas/atlas-lb/commit/d7d7f87.

[13] atlas/rax/api/validation/validator/raxloadbalancervalidatortest.java. https://github.com/openstack-atlas/atlas-lb/commit/3bc61ec.

[14] Chapter 14. blocks and statements. `https://docs.oracle.com/javase/specs/jls/se13/html/jls-14.html#jls-14.8`. (Accessed on 11/11/2019).

[15] Choreoswebserviceproxy/src/test/java/ime/usp/br/proxy/proxycontrollertest.java. `https://github.com/choreos/choreos_middleware/commit/f2da1f8`.

[16] Collections framework overview. `https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html`. (Accessed on 11/11/2019).

[17] core/internal/proportions/projectchangenotificationspdetest.java. `https://github.com/usus/usus-plugins/commit/0a66ccd`.

[18] core/src/main/java/org/mapfish/print/http/httprequestcache.java. `https://github.com/mapfish/mapfish-print/commit/fe44bd1`.

[19] core/src/main/java/org/mapfish/print/processor/map/createmapprocessor.java. `https://github.com/mapfish/mapfish-print/commit/bc1f422`.

[20] db/src/main/java/com/psddev/dari/db/sqldatabase.java. `https://github.com/perfectsense/dari/commit/88e6556`.

[21] de.prob.units/src/de/prob/units/sc/contextattributeprocessor.java. `https://github.com/hhu-stups/prob-rodinplugin/commit/32601b5`.

[22] domain/dependence/checkmethodtest.java. `https://github.com/socialsoftware/blended-workflow/commit/92a9539`.

[23] Getmetricstreamscopeidsdbmappertest.java. `https://github.com/lmco/eurekastreams/commit/9d456b6`.

[24] heroku-api/src/main/java/com/heroku/api/command/login/basicauthlogincommand.java. `https://github.com/heroku/heroku.jar/commit/008dbc2`.

[25] heroku-api/src/main/java/com/heroku/api/command/login/basicauthlogin.java. `https://github.com/heroku/heroku.jar/commit/0c1c18d`.

[26] hibernate-ogm-core/src/main/java/org/hibernate/ogm/grid/entitykey.java. `https://github.com/hibernate/hibernate-ogm/commit/7dcfaed`.

[27] io/searchbox/indices/deleteindexintegrationtest.java. `https://github.com/searchbox-io/jest/commit/0c5346a`.

[28] jack-core/src/com/rapleaf/jack/queries/generictable.java. https://github.com/liveramp/jack/commit/b331247.

[29] jack-store/src/com/rapleaf/jack/store/jstable.java. https://github.com/liveramp/jack/commit/762b540.

[30] jangaroo/jangaroo-compiler/src/main/java/net/jangaroo/jooc/jangarooparser.java. https://github.com/coremedia/jangaroo-tools/commit/7a494f1.

[31] jangaroo/jangaroo-compiler/src/main/java/net/jangaroo/jooc/jangarooparser.java. https://github.com/coremedia/jangaroo-tools/commit/fc54b3f.

[32] java/org/pitest/pitclipse/pitrunner/config/pitexecutionmodetest.java. https://github.com/pitest/pitclipse/commit/751b3d7.

[33] Javaparser. http://javaparser.org/.

[34] Junit. https://junit.org/.

[35] Lateralgm/org/lateralgm/file/gmstreamdecoder.java. https://github.com/ismavatar/lateralgm/commit/2d1bdaf.

[36] mes-core/mes-core-data/src/main/java/com/qcadoo/mes/core/data/internal/dataaccessserviceimpl.java. https://github.com/qcadoo/mes/commit/15a2615.

[37] name.abuchen.portfolio.ui/src/name/abuchen/portfolio/ui/util/bindinghelper.java. https://github.com/buchen/portfolio/commit/1bdeccb.

[38] nuget-tests/src/jetbrains/buildserver/nuget/tests/server/trigger/packagecheckertestbase.java. https://github.com/jetbrains/teamcity-nuget-support/commit/da10d2c.

[39] org.jrebirth.af/core/src/main/java/org/jrebirth/af/core/component/basic/innercomponentbase.java. https://github.com/jrebirth/jrebirth/commit/d82fb1b.

[40] org/lateralgm/file/gmstreamdecoder.java. https://github.com/ismavatar/lateralgm/commit/e41c4c5.

[41] Primitive data types. https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html. (Accessed on 11/11/2019).

[42] qtiworks-engine/src/main/java/uk/ac/ed/ph/qtiworks/domain/entities/candidateevent.java. https://github.com/davemckain/qtiworks/commit/0c924ab.

[43] qtiworks/web/controller/instructor/instructorassessmentmanagementcontroller.java. https://github.com/davemckain/qtiworks/commit/2a1f9df.

[44] qtiworks/web/controller/instructor/instructorassessmentmanagementcontroller.java. https://github.com/davemckain/qtiworks/commit/9cb51b2.

[45] Simplegemfirerepositoryintegrationtests.java. https://github.com/spring-projects/spring-data-gemfire/commit/15aae69.

[46] /src/com/eclipsesource/tabris/internal/ui/remotepagetest.java. https://github.com/eclipsesource/tabris/commit/df56d08.

[47] src/main/java/com/atomicleopard/webframework/view/json/jsonview.java. https://github.com/3wks/thundr/commit/9b02920.

[48] src/main/java/com/atomicleopard/webframework/view/json/jsonviewresult.java. https://github.com/3wks/thundr/commit/53aaf15.

[49] src/main/java/com/stripe/model/applepaydomain.java. https://github.com/stripe/stripe-java/commit/19d4d5a.

[50] src/main/java/com/stripe/model/threedsecure.java. https://github.com/stripe/stripe-java/commit/4fdadaf.

[51] src/main/java/org/atlasapi/application/applicationconfiguration.java. https://github.com/atlasapi/atlas-model/commit/4da9fc2.

[52] src/main/java/org/atlasapi/application/applicationconfiguration.java. https://github.com/atlasapi/atlas-model/commit/fc19c98.

[53] src/main/java/org/sql2o/query.java. https://github.com/aaberg/sql2o/commit/2f23b11.

[54] src/main/java/se/crafted/chrisb/ecocreature/drops/sources/abstractdropsource.java. https://github.com/mung3r/ecocreature/commit/42e5d9f.

[55] src/main/java/se/crafted/chrisb/ecocreature/drops/sources/abstractdropsource.java. https://github.com/mung3r/ecocreature/commit/3e2f216.

[56] src/test/java/com/github/koraktor/mavanagaiata/abstractgitmojotest.java. https://github.com/koraktor/mavanagaiata/commit/27f52ab.

[57] src/test/java/com/google/common/truth/iterablesubjecttest.java. `https://github.com/google/truth/commit/5de3d21`.

[58] src/test/java/com/stripe/model/issuerfraudrecordtest.java. `https://github.com/stripe/stripe-java/commit/42e8cec`.

[59] src/test/java/hudson/scm/subversionscmtest.java. `https://github.com/jenkinsci/subversion-plugin/commit/179fec8`.

[60] src/test/java/io/vertx/core/http/httptest.java. `https://github.com/eclipse-vertx/vert.x/commit/cdc8172`.

[61] src/test/java/io/vertx/test/core/fileresolvertest.java. `https://github.com/eclipse-vertx/vert.x/commit/f5bfd8c`.

[62] src/test/java/javax/cache/cachetest.java. `https://github.com/jsr107/jsr107tck/commit/27149d0`.

[63] src/test/java/joptsimple/internal/reflectiontest.java. `https://github.com/jopt-simple/jopt-simple/commit/cf6d097`.

[64] src/test/java/net/greghaines/jesque/integrationtest.java. `https://github.com/gresrun/jesque/commit/3f6a680`.

[65] src/test/java/net/sourceforge/pmd/lang/java/dfa/acceptancetest.java. `https://github.com/adangel/pmd/commit/e280151`.

[66] /src/test/java/org/apache/hama/bsp/message/testavromessagemanager.java. `https://github.com/apache/hama/commit/a5483d1`.

[67] src/test/java/org/jsr107/tck/cachetest.java. `https://github.com/jsr107/jsr107tck/commit/c2d9369`.

[68] src/test/java/org/junit/contrib/truth/collectiontest.java. `https://github.com/google/truth/commit/2b57a43`.

[69] src/test/java/org/linqs/psl/model/rule/groundruletest.java. `https://github.com/linqs/psl/commit/bab277d`.

[70] src/test/java/org/motechproject/ananya/kilkari/handlers/obdrequesthandlertest.java. `https://github.com/motech/ananya-kilkari/commit/b3b95f4`.

[71] src/test/java/org/neo4j/cypherdsl/cypherreferencetest.java. `https://github.com/neo4j-contrib/cypher-dsl/commit/4ae0202`.

[72] src/test/java/org/resthub/web/controller/testresourcecontroller.java. `https://github.com/resthub/resthub-spring-stack/commit/c55d122`.

[73] src/test/java/org/scribble/projection/protocolprojectiontest.java. `https://github.com/scribble/scribble-java/commit/62d9cdb`.

[74] src/test/java/stormpot/configtest.java. `https://github.com/chrisvest/stormpot/commit/b4eeba8`.

[75] src/test/java/stormpot/countingallocator.java. `https://github.com/chrisvest/stormpot/commit/459d423`.

[76] src/test/java/stormpot/countingallocatorwrapper.java. `https://github.com/chrisvest/stormpot/commit/d2931d3`.

[77] src/test/org/apache/hama/matrix/testsingularvaluedecomposition.java. `https://github.com/apache/hama/commit/6ba4dc3`.

[78] test-src/net/dougqh/jak/jvm/assembler/api/genericstest.java. `https://github.com/dougqh/jak/commit/7d4cba0`.

[79] test/de/jungblut/classification/nn/multilayerperceptrontest.java. `https://github.com/thomasjungblut/thomasjungblut-common/commit/2bce452`.

[80] test/georegression/geometry/testutilpoint3d_f64.java. `https://github.com/lessthanoptimal/georegression/commit/ed24838`.

[81] test/org/freenetproject/freemail/imap/imapuidfetchtest.java. `https://github.com/freenet/plugin-freemail/commit/778a842`.

[82] unittests/counterexampleuntilunittest.java. `https://github.com/hhu-stups/prob-rodinplugin/commit/c3f554b`.

[83] vertx-core/src/main/java/io/vertx/core/http/httpclientoptions.java. `https://github.com/eclipse-vertx/vert.x/commit/921c69e`.

[84] S. L. Abebe and P. Tonella. Automated identifier completion and replacement. In *2013 17th European Conference on Software Maintenance and Reengineering*, pages 263–272, March 2013.

[85] Surafel Lemma Abebe, Sonia Haiduc, Paolo Tonella, and Andrian Marcus. Lexicon bad smells in software. In *2009 16th Working Conference on Reverse Engineering*, pages 95–99, 2009.

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

[87] C. Albon. *Machine Learning with Python Cookbook: Practical Solutions from Preprocessing to Deep Learning*. O'Reilly Media, 2018.

[88] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Learning natural coding conventions. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014.

[89] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 38–49, New York, NY, USA, 2015. ACM.

[90] Miltiadis Allamanis and Charles Sutton. Mining source code repositories at massive scale using language modeling. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 207–216. IEEE Press, 2013.

[91] Eman Abdullah AlOmar, Anthony Peruma, Mohamed Wiem Mkaouer, Christian Newman, Ali Ouni, and Marouane Kessentini. How we refactor and how we document it? on the use of supervised machine learning algorithms to classify refactoring documentation. *Expert Systems with Applications*, 167:114176, 2021.

[92] Eman Abdullah Alomar, Anthony Peruma, Mohamed Wiem Mkaouer, Christian D. Newman, and Ali Ouni. Behind the scenes: On the relationship between developer experience and refactoring. *Journal of Software: Evolution and Process*, n/a(n/a):e2395.

[93] Eman Abdullah AlOmar, Anthony Peruma, Christian D. Newman, Mohamed Wiem Mkaouer, and Ali Ouni. An exploratory study on refactoring documentation in issues handling. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, MSR '22, 5 2022.

[94] R. S. Alsuhaibani, C. D. Newman, M. L. Collard, and J. I. Maletic. Heuristic-based part-of-speech tagging of source code identifiers and comments. In *2015 IEEE 5th Workshop on Mining Unstructured Data (MUD)*, pages 1–6, Sep. 2015.

[95] Reem Alsuhaibani, Christian Newman, Michael Decker, Michael Collard, and Jonathan Maletic. On the naming of methods: A survey of professional developers. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 587–599, 2021.

[96] Scott W. Ambler, Alan Vermeulen, and Greg Bumgardner. *The Elements of Java Style*. Cambridge University Press, USA, 1999.

[97] J. Andersen, A. C. Nguyen, D. Lo, J. L. Lawall, and S. C. Khoo. Semantic patch inference. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 382–385, Sept 2012.

[98] Apache-Cocoon. cocoon/simplecocooncrawlerimpl.java. `https://github.com/apache/cocoon/blob/3625cf736722a4f501111398bfe65aadf534e69c/core/cocoon-core/src/main/java/org/apache/cocoon/components/crawler/SimpleCocoonCrawlerImpl.java`. (Accessed on 04/06/2021).

[99] Apache-Cocoon. cocoon/xmlbytestreamfragment.java. `https://github.com/apache/cocoon/blob/3625cf736722a4f501111398bfe65aadf534e69c/core/cocoon-pipeline/cocoon-pipeline-impl/src/main/java/org/apache/cocoon/components/sax/XMLByteStreamFragment.java`. (Accessed on 04/06/2021).

[100] A. April and A. Abran. *Software Maintenance Management: Evaluation and Continuous Improvement*. Practitioners. Wiley, 2012.

[101] V. Arnaoudova, L. M. Eshkevari, M. D. Penta, R. Oliveto, G. Antoniol, and Y. Guéhéneuc. Repent: Analyzing the nature of identifier renamings. *IEEE Transactions on Software Engineering*, 40(5):502–532, 2014.

[102] Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. Linguistic antipatterns: what they are and how developers perceive them. *Empirical Software Engineering*, 21(1):104–158, Feb 2016.

[103] Venera Arnaoudova, Massimiliano Di Penta, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. A new family of software anti-patterns: Linguistic anti-patterns. In *2013 17th European Conference on Software Maintenance and Reengineering*, pages 187–196, 2013.

[104] Anton Barua, Stephen W. Thomas, and Ahmed E. Hassan. What are developers talking about? an analysis of topics and trends in stack overflow. *Empirical Software Engineering*, 19(3):619–654, Jun 2014.

[105] Gal Beniamini, Sarah Gingichashvili, Alon Klein Orbach, and Dror G. Feitelson. Meaningful identifier names: The case of single-letter variables. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 45–54, 2017.

[106] Dave Binkley, Matthew Hearn, and Dawn Lawrie. Improving identifier informativeness using part of speech information. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, page 203–206, New York, NY, USA, 2011. Association for Computing Machinery.

[107] Dave Binkley, Dawn Lawrie, and Christopher Morrell. The need for software specific natural language techniques. *Empirical Softw. Engg.*, 23(4):2398–2425, August 2018.

[108] S. Bird, E. Klein, and E. Loper. *Natural Language Processing with Python: Analyzing Text with the Natural Language Toolkit*. O'Reilly Media, 2009.

[109] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003.

[110] B. W. Boehm. Software engineering economics. *IEEE Transactions on Software Engineering*, SE-10(1):4–21, Jan 1984.

[111] Leo Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, October 2001.

[112] R. P. L. Buse and W. R. Weimer. Learning a metric for code readability. *IEEE Transactions on Software Engineering*, 36(4):546–558, 2010.

[113] S. Butler, M. Wermelinger, and Y. Yu. A survey of the forms of java reference names. In *2015 IEEE 23rd International Conference on Program Comprehension*, pages 196–206, May 2015.

[114] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp. Relating identifier naming flaws and code quality: An empirical study. In *2009 16th Working Conference on Reverse Engineering*, pages 31–35, Oct 2009.

[115] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp. Mining java class naming conventions. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 93–102, Sep. 2011.

[116] Simon Butler. The effect of identifier naming on source code readability and quality. In *Proceedings of the Doctoral Symposium for ESEC/FSE on Doctoral Symposium*, ESEC/FSE Doctoral Symposium '09, page 33–34, New York, NY, USA, 2009. Association for Computing Machinery.

[117] Simon Butler, Michel Wermelinger, and Yijun Yu. Investigating naming convention adherence in java references. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 41–50, 2015.

[118] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. Exploring the influence of identifier names on code quality: An empirical study. In *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*, pages 156–165. IEEE, 2010.

[119] M. L. Collard and J. I. Maletic. srcml 1.0: Explore, analyze, and manipulate source code. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 649–649, Oct 2016.

[120] Michael L. Collard, Michael John Decker, and Jonathan I. Maletic. Srcml: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, ICSM '13, page 516–519, USA, 2013. IEEE Computer Society.

[121] T. A. Corbi. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1989.

[122] D.A. Cruse. Lexical semantics. In Neil J. Smelser and Paul B. Baltes, editors, *International Encyclopedia of the Social & Behavioral Sciences*, pages 8758–8764. Pergamon, Oxford, 2001.

[123] P. Dangeti. *Statistics for Machine Learning*. Packt Publishing, 2017.

[124] Florian Deissenboeck and Markus Pizka. Concise and consistent naming. *Software Quality Journal*, 14(3):261–282, Sep 2006.

[125] Florian Deissenboeck and Markus Pizka. Concise and consistent naming. *Software Quality Journal*, 14(3):261–282, September 2006.

[126] F. Dell'Orletta. Ensemble system for part-of-speech tagging. In *Proceedings of EVALITA 2009*, 2009.

[127] Natalia Dragan, Michael L. Collard, and Jonathan I. Maletic. Reverse engineering method stereotypes. In *Proceedings of the 22Nd IEEE International Conference on Software Maintenance*, ICSM '06, pages 24–34, Washington, DC, USA, 2006. IEEE Computer Society.

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

[129] Sarah Fakhoury, Yuzhan Ma, Venera Arnaoudova, and Olusola Adesope. The effect of poor source code lexicon and readability on developers' cognitive load. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pages 286–28610, 2018.

[130] Sarah Fakhoury, Devjeet Roy, Sk. Adnan Hassan, and Venera Arnaoudova. Improving source code readability: Theory and practice. In *Proceedings of the 27th International Conference on Program Comprehension*, ICPC '19, pages 2–12, Piscataway, NJ, USA, 2019. IEEE Press.

[131] Dror Feitelson, Ayelet Mizrahi, Nofar Noy, Aviad Ben Shabat, Or Eliyahu, and Roy Sheffer. How developers choose names. *IEEE Transactions on Software Engineering*, pages 1–1, 2020.

[132] Maryanne Fisher, Anthony Cox, and Lin Zhao. Using sex differences to link spatial cognition and program comprehension. In *Proceedings of the 22Nd IEEE International Conference on Software Maintenance*, ICSM '06, pages 289–298, Washington, DC, USA, 2006. IEEE Computer Society.

[133] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Signature Series (Fowler). Pearson Education, 2018.

[134] D. Galin. *Software Quality: Concepts and Practice.* Wiley, 2018.

[135] P. Goodliffe. *Code Craft: The Practice of Writing Excellent Code.* No Starch Press Series. No Starch Press, 2007.

[136] Google. Java style guide. `https://google.github.io/styleguide/javaguide.html`. (Accessed on 03/25/2021).

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

[138] S. Haiduc, J. Aponte, and A. Marcus. Supporting program comprehension with source code summarization. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 2, pages 223–226, May 2010.

[139] E. Hill, L. Pollock, and K. Vijay-Shanker. Automatically capturing source code context of nl-queries for software maintenance and reuse. In *2009 IEEE 31st International Conference on Software Engineering*, pages 232–242, May 2009.

[140] E. Hill, L. Pollock, and K. Vijay-Shanker. Improving source code search with natural language phrasal representations of method signatures. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 524–527, Nov 2011.

[141] Emily Hill. *Integrating Natural Language and Program Structure Information to Improve Software Search and Exploration.* PhD thesis, Newark, DE, USA, 2010. AAI3423409.

[142] Emily Hill, David Binkley, Dawn Lawrie, Lori Pollock, and K. Vijay-Shanker. An empirical study of identifier splitting techniques. *Empirical Softw. Engg.*, 19(6):1754–1780, December 2014.

[143] Emily Hill, Zachary P. Fry, Haley Boyd, Giriprasad Sridhara, Yana Novikova, Lori Pollock, and K. Vijay-Shanker. Amap: Automatically mining abbreviation expansions in programs to enhance software maintenance tools. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, MSR '08, pages 79–88, New York, NY, USA, 2008. ACM.

[144] J. Hofmeister, J. Siegmund, and D. V. Holt. Shorter identifier names take longer to comprehend. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 217–227, 2017.

[145] Einar W. Høst and Bjarte M. Østvold. Debugging method names. In Sophia Drossopoulou, editor, *ECOOP 2009 – Object-Oriented Programming*, pages 294–317, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[146] Michael Hucka. Spiral: splitters for identifiers in source code files. *Journal of Open Source Software*, 3(24):653, 2018.

[147] L. Jiang, H. Liu, and H. Jiang. Machine learning based recommendation of method names: How far are we. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 602–614, 2019.

[148] Yanjie Jiang, Hui Liu, Jiahao Jin, and Lu Zhang. Automated expansion of abbreviations based on semantic relation and transfer expansion. *IEEE Transactions on Software Engineering*, 48(2):519–537, 2022.

[149] Yanjie Jiang, Hui Liu, Yuxia Zhang, Nan Niu, Yuhai Zhao, and Lu Zhang. Which abbreviations should be expanded? In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 578–589, New York, NY, USA, 2021. Association for Computing Machinery.

[150] Robbert Jongeling, Proshanta Sarkar, Subhajit Datta, and Alexander Serebrenik. On negative results when using sentiment analysis tools for software engineering research. *Empirical Software Engineering*, 01 2017.

[151] Daniel Jurafsky and James H Martin. Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition. *Prentic e Hall*, 2019.

[152] Y. Kashiwabara, Y. Onizuka, T. Ishio, Y. Hayase, T. Yamamoto, and K. Inoue. Recommending verbs for rename method using association rule mining. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 323–327, Feb 2014.

[153] D. E. Krutz, N. Munaiah, A. Peruma, and M. Wiem Mkaouer. Who added that permission to my app? an analysis of developer permission changes in open source android apps. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 165–169, May 2017.

[154] D. Lawrie and D. Binkley. Expanding identifiers to normalize source code vocabulary. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 113–122, Sept 2011.

[155] D. Lawrie, C. Morrell, H. Feild, and D. Binkley. What's in a name? a study of identifiers. In *14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 3–12, 2006.

[156] Huiqing Li and Simon Thompson. Let's make refactoring tools user-extensible! In *Proceedings of the Fifth Workshop on Refactoring Tools*, WRT '12, page 32–39, New York, NY, USA, 2012. Association for Computing Machinery.

[157] Ben Liblit, Andrew Begel, and Eve Sweetser. Cognitive perspectives on the role of naming in computer programs. In *In Proc. of the 18th Annual Psychology of Programming Workshop*, 2006.

[158] B. P. Lientz, E. B. Swanson, and G. E. Tompkins. Characteristics of application software maintenance. *Commun. ACM*, 21(6):466–471, June 1978.

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

[160] Hui Liu, Qiurong Liu, Cristian-Alexandru Staicu, Michael Pradel, and Yue Luo. Nomen est omen: Exploring and exploiting similarities between argument and parameter names. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 1063–1073, New York, NY, USA, 2016. Association for Computing Machinery.

[161] Kui Liu, Dongsun Kim, Tegawendé F. Bissyandé, Taeyoung Kim, Kisub Kim, Anil Koyuncu, Suntae Kim, and Yves Le Traon. Learning to spot and refactor inconsistent method names. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE 2019, New York, NY, USA, 2019. ACM.

[162] Walid Maalej, Rebecca Tiarks, Tobias Roehm, and Rainer Koschke. On the comprehension of program comprehension. *ACM Trans. Softw. Eng. Methodol.*, 23(4), September 2014.

[163] Christopher D. Manning and Hinrich Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, MA, USA, 1999.

[164] R.C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Robert C. Martin series. Prentice Hall, 2009.

[165] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1 edition, 2008.

[166] Na Meng, Miryung Kim, and Kathryn S. McKinley. Lase: Locating and applying systematic edits by learning from examples. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 502–511, Piscataway, NJ, USA, 2013. IEEE Press.

[167] Microsoft. C# coding conventions. `https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/inside-a-program/coding-conventions`. (Accessed on 03/25/2021).

[168] George A. Miller. Wordnet: A lexical database for english. *Commun. ACM*, 38(11):39–41, November 1995.

[169] MinecraftForge-FML. client/cpw/mods/fml/client/guislotmodlist.java. `https://github.com/MinecraftForge/FML/commit/72edbe596e3c02a2c08f818572140f955ea43112#diff-b2e2a64fb208ef8ea08d5db83de1f4337c732fb4c5fdf5dc5450da94934d6d44`. (Accessed on 04/06/2021).

[170] Maddeh Mohamed, Mohamed Romdhani, and Khaled Ghédira. Classification of model refactoring approaches. *Journal of Object Technology*, 8(6):121–126, 2009.

[171] Dan H. Moore II. Classification and regression trees, by leo breiman, jerome h. friedman, richard a. olshen, and charles j. stone. brooks/cole publishing, monterey, 1984,358 pages, $27.95. *Cytometry*, 8(5):534–535, 1987.

[172] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. Curating github for engineered software projects. *Empirical Software Engineering*, 22(6):3219–3253, Dec 2017.

[173] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18, Jan 2012.

[174] C. Newman, M. J. Decker, R. S. AlSuhaibani, A. Peruma, D. Kaushik, and E. Hill. An open dataset of abbreviations and expansions. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 280–280, 2019.

[175] C. D. Newman, R. S. AlSuhaibani, M. L. Collard, and J. I. Maletic. Lexical categories for source code identifiers. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 228–239, Feb 2017.

[176] C. D. Newman, M. J. Decker, R. S. Alsuhaibani, A. Peruma, D. Kaushik, and E. Hill. An empirical study of abbreviations and expansions in software artifacts. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 269–279, 2019.

[177] Christian D. Newman, Reem S. AlSuhaibani, Michael J. Decker, Anthony Peruma, Dishant Kaushik, Mohamed Wiem Mkaouer, and Emily Hill. On the generation, structure, and semantics of grammar patterns in source code identifiers. *Journal of Systems and Software*, 170, Dec 2020.

[178] Christian D. Newman, Michael J Decker, Reem Alsuhaibani, Anthony Peruma, Mohamed Mkaouer, Satyajit Mohapatra, Tejal Vishoi, Marcos Zampieri, Timothy Sheldon, and Emily Hill. An ensemble approach for annotating source code identifiers with part-of-speech tags. *IEEE Transactions on Software Engineering*, pages 1–1, 2021.

[179] Christian D. Newman, Anthony Preuma, and Reem AlSuhaibani. Modeling the relationship between identifier name and behavior. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 376–378, 2019.

[180] W. Olney, E. Hill, C. Thurber, and B. Lemma. Part of speech tagging java method names. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 483–487, Oct 2016.

[181] J. Pantiuchina, M. Lanza, and G. Bavota. Improving code: The (mis) perception of quality metrics. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018.

[182] Jevgenija Pantiuchina, Fiorella Zampetti, Simone Scalabrino, Valentina Piantadosi, Rocco Oliveto, Gabriele Bavota, and Massimiliano Di Penta. Why developers refactor source code: A mining-based study. *ACM Trans. Softw. Eng. Methodol.*, 29(4), September 2020.

[183] Profir-Petru Pârtachi, Santanu Kumar Dash, Christoph Treude, and Earl T. Barr. Posit: Simultaneously tagging natural and programming languages. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 1348–1358, New York, NY, USA, 2020. Association for Computing Machinery.

[184] A. Peruma. A preliminary study of android refactorings. In *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 148–149, 2019.

[185] A. Peruma, M. W. Mkaouer, M. J. Decker, and C. D. Newman. Contextualizing rename decisions using refactorings and commit messages. In *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2019.

[186] Anthony Peruma. Towards a model to appraise and suggest identifier names. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 639–643, 09 2019.

[187] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. On the distribution of test smells in open source android applications: An exploratory study. In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, CASCON '19, page 193–202, USA, 2019. IBM Corp.

[188] Anthony Peruma, Eman Abdullah AlOmar, Christian D. Newman, Mohamed Wiem Mkaouer, and Ali Ouni. Refactoring debt: Myth or reality? an exploratory study on the relationship between technical debt and refactoring. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, MSR '22, 5 2022.

[189] Anthony Peruma, Venera Arnaoudova, and Christian D. Newman. Ideal: An open-source identifier name appraisal tool. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 599–603, 2021.

[190] Anthony Peruma, Emily Hu, Jiajun Chen, Eman Abdullah Alomar, Mohamed Wiem Mkaouer, and Christian D. Newman. Using grammar patterns to interpret test method name evolution. In *Proceedings of the 29th International Conference on Program Comprehension*, ICPC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[191] Anthony Peruma, Mohamed Wiem Mkaouer, Michael J. Decker, and Christian D. Newman. An empirical investigation of how and why developers rename identifiers. In *Proceedings of the 2nd International Workshop on Refactoring*, IWoR, 2018.

[192] Anthony Peruma, Mohamed Wiem Mkaouer, Michael J. Decker, and Christian D. Newman. Contextualizing rename decisions using refactorings, commit messages, and data types. *Journal of Systems and Software*, 169, 2020.

[193] Anthony Peruma and Christian D. Newman. On the distribution of "simple stupid bugs" in unit test files: An exploratory study. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 525–529, 2021.

[194] Anthony Peruma and Christian D. Newman. Understanding digits in identifier names: An exploratory study. In *The 1st International Workshop on Natural Language-based Software Engineering (NLBSE)*, NLBSE '22, 5 2022.

[195] Anthony Peruma, Christian D. Newman, Mohamed Wiem Mkaouer, Ali. Ouni, and Fabio Palomba. An exploratory study on the refactoring of unit test files in android applications. In *Proceedings of the 4th International Workshop on Refactoring*, New York, NY, USA, 06 2020.

[196] Anthony Peruma, Steven Simmons, Eman Abdullah AlOmar, Christian D. Newman, Mohamed Wiem Mkaouer, and Ali Ouni. How do i refactor this? an empirical study on refactoring trends and topics in stack overflow. *Empirical Software Engineering*, 27(1):11, Oct 2021.

[197] Daryl Posnett, Abram Hindle, and Premkumar Devanbu. A simpler model of software readability. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, page 73–82, New York, NY, USA, 2011. Association for Computing Machinery.

[198] R.S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill higher education. McGraw-Hill Education, 2010.

[199] V. Rajich. Program comprehension as a learning process. In *Proceedings First IEEE International Conference on Cognitive Informatics*, pages 343–347, 2002.

[200] V. Rajlich and N. Wilde. The role of concepts in program comprehension. In *Proceedings 10th International Workshop on Program Comprehension*, pages 271–278, 2002.

[201] Radim Řehůřek and Petr Sojka. Software framework for topic modelling with large corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010. ELRA. `http://is.muni.cz/publication/884893/en`.

[202] Michael Röder, Andreas Both, and Alexander Hinneburg. Exploring the space of topic coherence measures. In *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining*, WSDM '15, pages 399–408, New York, NY, USA, 2015. ACM.

[203] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann. Learning syntactic program transformations from examples. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 404–415, May 2017.

[204] Devjeet Roy, Sarah Fakhoury, John Lee, and Venera Arnaoudova. A model to detect readability improvements in incremental changes. In *Proceedings of the 28th International Conference on Program Comprehension*, ICPC '20, page 25–36, New York, NY, USA, 2020. Association for Computing Machinery.

[205] Simone Scalabrino, Mario Linares-Vásquez, Rocco Oliveto, and Denys Poshyvanyk. A comprehensive model for code readability. *Journal of Software: Evolution and Process*, 30(6):e1958, 2018. e1958 smr.1958.

[206] A. Schankin, A. Berger, D. V. Holt, J. C. Hofmeister, T. Riedel, and M. Beigl. Descriptive compound identifier names improve source code comprehension. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pages 31–3109, 2018.

[207] Bonita Sharif and Jonathan I. Maletic. An eye tracking study on camelcase and under_score identifier styles. In *2010 IEEE 18th International Conference on Program Comprehension*, pages 196–205, 2010.

[208] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. Why we refactor? confessions of github contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016. Association for Computing Machinery, 2016.

[209] Charles Simonyi and Martin Heller. The hungarian revolution. *BYTE*, 16(8):131–ff., August 1991.

[210] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, SE-10(5):595–609, 1984.

[211] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 43–52, New York, NY, USA, 2010. ACM.

[212] Takayuki Suzuki, Kazunori Sakamoto, Fuyuki Ishikawa, and Shinichi Honiden. An approach for evaluating and suggesting method names using n-gram models. In *Proceedings of the 22nd International Conference on Program Comprehension*, ICPC 2014, page 271–274, New York, NY, USA, 2014. Association for Computing Machinery.

[213] Armstrong A. Takang, Penny A. Grubb, and Robert D. Macredie. The effects of comments and identifier names on program comprehensibility: an experimental investigation. *J. Prog. Lang.*, 4:143–167, 1996.

[214] Liang Tan and Christoph Bockisch. A survey of refactoring detection tools. In *Software Engineering*, 2019.

[215] A. Tashakkori, C. Teddlie, and C.B. Teddlie. *Mixed Methodology: Combining Qualitative and Quantitative Approaches*. Applied Social Research Methods. SAGE Publications, 1998.

[216] Kristina Toutanova, Dan Klein, Christopher D Manning, and Yoram Singer. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings of the 2003 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics*, pages 252–259, 2003.

[217] Kristina Toutanova and Christopher D. Manning. Enriching the knowledge sources used in a maximum entropy part-of-speech tagger. In *Proceedings of the 2000 Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora: Held in Conjunction with the 38th Annual Meeting of the Association for Computational Linguistics - Volume 13*, EMNLP '00, pages 63–70, Stroudsburg, PA, USA, 2000. Association for Computational Linguistics.

[218] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 483–494, New York, NY, USA, 2018. Association for Computing Machinery.

[219] Carmine Vassallo, Giovanni Grano, Fabio Palomba, Harald C Gall, and Alberto Bacchelli. A large-scale empirical exploration on refactoring activities in open source software projects. *Science of Computer Programming*, 180, 2019.

[220] A. Von Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995.

[221] Anneliese von Mayrhauser and A. Marie Vans. Program understanding behavior during debugging of large scale software. In *Papers Presented at the Seventh Workshop on Empirical Studies of Programmers*, ESP '97, pages 157–179, New York, NY, USA, 1997. ACM.

[222] Jianwei Wu and James Clause. A pattern-based approach to detect and improve non-descriptive test names. *Journal of Systems and Software*, 168:110639, 2020.

[223] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering*, 44(10):951–976, 2018.

[224] Z. Xing and E. Stroulia. Refactoring detection based on umldiff change-facts queries. In *2006 13th Working Conference on Reverse Engineering*, pages 263–274, Oct 2006.

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

[226] Benwen Zhang, Emily Hill, and James Clause. Towards automatically generating descriptive names for unit tests. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, page 625–636, New York, NY, USA, 2016. Association for Computing Machinery.

[227] J. Zhang, S. Liu, J. Luo, J. Liang, and Z. Huang. Exploring the characteristics of identifiers: A large-scale empirical study on 5,000 open source projects. *IEEE Access*, 8:140607–140620, 2020.

Appendices

Appendix A

Publication List

Table A.1 depicts the list of peer-reviewed scientific publications, related to my primary research topic, I have authored during my PhD program.

Table A.1: List of identifier name related publications

Year	Venue	Title	Reference
2018	IWoR	An Empirical Investigation of How and Why Developers Rename Identifiers	[191]
2019	MobileSoft	A Preliminary Study of Android Refactorings	[184]
2019	ICSME	An Empirical Study of Abbreviations and Expansions in Software Artifacts	[176]
2019	ICSME	An Open Dataset of Abbreviations and Expansions	[174]
2019	SCAM	Contextualizing Rename Decisions using Refactorings and Commit Messages	[185]
2019	ICSME	Modeling the Relationship Between Identifier Name and Behavior	[179]
2019	ICSME	Towards a Model to Appraise and Suggest Identifier Names	[186]
2020	JSS	Contextualizing rename decisions using refactorings, commit messages, and data types	[192]
2020	JSS	On the generation, structure, and semantics of grammar patterns in source code identifiers	[177]
2021	JSEP	Behind the scenes: On the relationship between developer experience and refactoring	[92]
2021	ICPC	Using Grammar Patterns to Interpret Test Method Name Evolution	[190]
2021	MSR	On the Distribution of "Simple Stupid Bugs" in Unit Test Files: An Exploratory Study	[193]
2021	TSE	An Ensemble Approach for Annotating Source Code Identifiers with Part-of-speech Tags	[178]
2021	ICSME	IDEAL: An Open-Source Identifier Name Appraisal Tool	[189]
2021	ESEM	How Do I Refactor This? An Empirical Study on Refactoring Trends and Topics in Stack Overflow	[196]
2022	NLBSE	Understanding Digits in Identifier Names: An Exploratory Study	[194]
2022	MSR	An Exploratory Study on Refactoring Documentation in Issues Handling	[93]
2022	MSR	Refactoring Debt: Myth or Reality? An Exploratory Study on the Relationship Between Technical Debt and Refactoring	[188]

Appendix B

User Study Questionnaires

The contents of this appendix are the questionnaires that were completed by the participants in the user study for the IDE name recommendation and appraisal plugin (refer to Chapter 11). Participants had to complete a pre- and post-questionnaire.

B.1 Plugin Participant's Questionnaire

This section contains the pre- and post-questionnaire provided to participants who utilized the plugin.

PRE QUESTIONNAIRE

1. **Academic Program**:
   [select one] Bachelor ; Master ; PhD ; Other____

2. **Academic Major:**

3. **General Programming Experience (Years)**:

4. **Java Programming Experience (Years)**:

5. **IntelliJ IDEA Hands-on Familiarity**:
   [select one] Extremely ; Very ; Moderately ; Slightly ; Not at all

6. **A tool providing me with real-time grammar pattern-based naming suggestions for identifier names would be _________:**
   [select one] Very convenient ; Convenient ; Neutral ; Inconvenient ; Very inconvenient

———end———

POST QUESTIONNAIRE

1. **Did you execute/run the code snippets to better understand the purpose of the program?**
   [select one] Yes ; No
   a. **What code snippets did you run?**

2. **What is the purpose of the following functions in the file “StringUtilities.java”:**
   a. `public static boolean letters(String stringsToTest)`
   b. `public static int counter(String words)`

3. **In the file “StringUtilities.java”, you were explicitly asked to rename two identifiers (line 5 & line 20). What rationale did you use in selecting the replacement names?**
   a. Line 5:
   b. Line 20:

4. **What were some common naming violations you experienced in the code snippets?**

5. **When crafting identifier names, how much of a priority did you place on the part-of-speech tags of the name?**
   [select one] Not a priority ; Low priority ; Medium priority ; High priority ; Essential
   a. **Please let us know why you selected the above option:**

6. **A tool providing me with real-time grammar pattern-based naming suggestions for identifier names was _______:**
   [select one] Very convenient ; Convenient ; Neutral ; Inconvenient ; Very inconvenient
   a. **Please let us know why you selected the above option:**

7. **Is the sidebar with naming recommendations easy to locate?**
   [select one] Yes ; No
   a. **If you selected "*No*" please let us know why:**

8. **Interpreting the primary recommendations proposed by the plugin was:**
   [select one] Very difficult ; Difficult ; Neutral ; Easy ; Very easy
   b. **Please let us know why you selected the above option:**

9. **Interpreting the other possible patterns proposed by the plugin was:**
   [select one] Very difficult ; Difficult ; Neutral ; Easy ; Very easy
   a. **Please let us know why you selected the above option:**

10. **Please rate your level of satisfaction with the accuracy of the recommendations:**
    [select one] Very dissatisfied ; Dissatisfied ; Unsure ; Satisfied ; Very satisfied
    a. **Please let us know why you selected the above option:**

11. **What existing functionality or new features would you like to see improved/incorporated into the plugin? This includes features that enhance the user experience and correctness/accuracy. Please elaborate on specific issues; examples would be helpful.**

12. **Is there anything else you would like to let us know about this study?**

———end———

B.2 Non-Plugin Participant's Questionnaire

This section contains the pre- and post-questionnaire provided to participants who did not utilize the plugin.

PRE QUESTIONNAIRE

1. **Academic Program**:
   [select one] Bachelor ; Master ; PhD ; Other____

2. **Academic Major:**

3. **General Programming Experience (Years)**:

4. **Java Programming Experience (Years)**:

5. **IntelliJ IDEA Hands-on Familiarity**:
   [select one] Extremely ; Very ; Moderately ; Slightly ; Not at all

———*end*———

POST QUESTIONNAIRE

1. **Did you execute/run the code snippets to better understand the purpose of the program?**
   [select one] Yes ; No
   a. **What code snippets did you run?**

2. **What is the purpose of the following functions in the file "StringUtilities.java":**
   a. `public static boolean letters(String stringsToTest)`
   b. `public static int counter(String words)`

3. **In the file "StringUtilities.java", you were explicitly asked to rename two identifiers (line 5 & line 20). What rationale did you use in selecting the replacement names?**
   a. Line 5:
   b. Line 20:

4. **What were some common naming violations you experienced in the code snippets?**

5. **When crafting identifier names, how much of a priority do you place on the part-of-speech tags of the name?**
   [select one] Not a priority ; Low priority ; Medium priority ; High priority ; Essential
   a. **Please let us know why you selected the above option:**

6. **How do you rate the difficulty level of creating better identifier names?**

   [select one] Very difficult ; Difficult ; Neutral ; Easy ; Very easy

7. **Would you have liked to have a tool to recommend changes to make to identifier names?**

   [select one] Yes ; No

   a. **If you selected "*__No__*" please let us know why:**
   b. **If you selected "*__Yes__*" please let us know what key features the tool should have:**

8. **Is there anything else you would like to let us know about this study?**

———*end*———
