An Empirical Investigation of How and Why Developers Rename Identifiers

Anthony Peruma
Rochester Institute of Technology
Rochester, New York, United States
axp6201@rit.edu

Mohamed Wiem Mkaouer
Rochester Institute of Technology
Rochester, New York, United States
mwmvse@rit.edu

Michael J. Decker
Bowling Green State University
Bowling Green, Ohio, United States
mdecke@bgsu.edu

Christian D. Newman
Rochester Institute of Technology
Rochester, New York, United States
cnewman@se.rit.edu

ABSTRACT

Renaming is vital to software maintenance and evolution. Developers rename entities when their original names no longer fit their behavioral role within the program. This may happen if the entity's original name was of poor quality or if the system has evolved such that the original name needs to be updated to reflect some of this evolution. In the end, the reason for the rename ultimately falls under increasing understandability and comprehension. Because comprehension is so important, and identifiers are the primary way developers comprehend code, it is critical to understand empirically how and why identifier names evolve. Armed with an understanding of these two facets of identifier evolution, researchers will be able to train algorithms to recognize, recommend, or even automatically generate high-quality identifier names. We present an empirical study of how method, class and package identifier names evolve to better understand the motives of their evolution. The empirical validation involves a set of 524,113 rename refactorings, performed on 3,795 Java systems. In a nutshell, our findings demonstrate that most rename refactorings narrow the meaning of the identifiers for which they are applied. Further, we analyze commit messages to contextualize these renames.

CCS CONCEPTS

• **General and reference** → **Empirical studies**;

KEYWORDS

rename refactoring, natural language processing, program comprehension

ACM Reference Format:
Anthony Peruma, Mohamed Wiem Mkaouer, Michael J. Decker, and Christian D. Newman. 2018. An Empirical Investigation of How and Why Developers Rename Identifiers . In *Proceedings of the 2nd International Workshop on Refactoring (IwoR '18), September 4, 2018, Montpellier, France.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3242163.3242169

IwoR '18, September 4, 2018, Montpellier, France

ACM ISBN 978-1-4503-5974-0/18/09…$15.00
https://doi.org/10.1145/3242163.3242169

1 INTRODUCTION

The majority of software life-cycle resources are allocated to program maintenance [12, 15]. 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 [14]. 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 [1]. 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 [16, 17, 31, 32]. Furthermore, many previous studies have investigated naming practices, patterns, and how to improve analysis of identifier names [9, 10, 18, 21, 27]. In particular, a number of papers explore the idea of debugging, appraising, and generating identifier names [2, 4, 19]. These have a direct, positive impact on approaches that synthesize programs [6, 25, 29], 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 paper, 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. [8] 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. Similarly to Arnaoudova et al. [8], the first three research questions are primarily defined to explore our findings about 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 a first step towards better supporting tools that try to understand developer behavior when it comes to naming. This will eventually help increase adoption of technology that supports identifier name evolution and the creation of better guidelines for how renames should be applied and supported during maintenance.

2 ANALYSIS OF RENAMES

Arnaoudova et al [8] created a taxonomy of operations carried out during rename refactorings. We use this taxonomy to examine rename refactorings and categorize them into the different types prescribed by this taxonomy. In this section, we will briefly discuss the taxonomy, but encourage the reader to read the original work for a more thorough discussion of each category. The taxonomy is made up of five high-level categories which are presented below.

2.1 Taxonomy for Rename Refactorings

Entity Kind: This category is concerned with what source code entity a given identifier represents. For example, the identifier may be the name of a type, class, getter, setter, etc.

Form of Renaming: This category is interested in the lexical change made to the identifier. They break this down into a few subcategories: Simple, complex, reordering, and formatting. Simple changes are those that only add, remove, or change one term in the identifier; complex are those that add, remove or change multiple terms. Reordering is where two or more terms in an identifier switch positions (i.e., GetSetter becomes SetterGet), and formatting changes are those where there is no renaming but a letter in a term changes case or a separator (e.g., underscore) is added or removed.

Semantic Changes: These are changes due to adding/removing terms or modifying terms (e.g., to another term that is a synonym of the original) such that the meaning of the identifier may have been modified. The following heuristics are used to figure out whether the idenifier's semantics have been preserved or modified.

We consider **meaning 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 meaning of the identifier (i.e., ItemNotVisible becomes ItemHidden).

We consider **meaning modified** if one of the following holds: 1) *Generalize 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., GetFirstUnit becomes GetUnit). 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., GetUnit becomes GetFirstUnit). 3) We consider *meaning changed* (i.e., not narrowed or generalized) when an old term is changed to a new term which 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* via the addition of one or more new term to the identifier AND the addition does not fall into one of the categories above (e.g., narrow meaning). 5) *Remove meaning* via removal of

one or more terms from the identifier AND the removal does not fall into one of the categories above (e.g., generalize meaning).

Grammar Changes: These are modifications which cause the part of speech categorization of a term to change. For example, if FireCannon() becomes CannonFired() then *fire* changed from a verb to an adjective.

There is also a none category, for when changes did not fall into any of the above categories but also did not imply a modification of meaning. Examples of these are formatting changes, change in case, etc.

2.2 Contextualizing Rename Refactorings

Developers rename identifiers for multiple reasons. Through careful analysis of rename refactorings, one can gain insight into how developers choose their words, why they choose certain types of words over others, and how to mimic this process automatically. In this subsection, we show examples of how developer activity recorded in commit messages is reflected in their rename choices.

By analyzing the following method rename: *setDisableBinLogCache* → *setEnableReplicationCache*, we observe 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"* 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"*.

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 correcting 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"*. 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..."*, where renaming to 'proper English' indicates that the meaning has not been modified but should now be easier to comprehend.

Of course, examining every single commit message is infeasible due to the sheer number of them in the set of rename refactorings used in this study. Therefore, we use a topic modeling algorithm called Latent Dirichlet Allocation (LDA) to help us understand the main topics in a large set of documents (commit messages). The aim is to use words in commit messages (such as those above) to identify the reason behind different semantic changes.

3 RELATED WORK

Since the choice of adequate naming for identifiers is critical for code understandability [13], several studies have focused on suggesting more meaningful names for identifiers when identifying poorly named code elements. Liblit et al. [22] discusses naming in several programming languages and makes observations about how natural language influences the use of words in these languages. Høst et al. [19] designed automated naming rules using method signature elements, i.e., return type, parameters names and types, and control flow. They mined several projects and identified naming bugs, i.e., violations of their naming rules. Similarly, Arnaoudova et al. [9] defined a catalog of linguistic anti-patterns that are found to deteriorate the quality of code understanding. They have shown the negative impact of linguistic anti-patterns by conducting two empirical studies with software developers and finding that the majority of programmers perceive anti-patterns as poor naming practices [7].

Arnaoudova et al. [8] proposed REPENT, an approach to analyze and classify identifier renamings. They mined several rename attribute operations and then contrasted between the old and new namings using the lexical database Wordnet [26]. The authors have shown the impact of proper naming on minimizing software development effort.

Allamanis et al. [3] used statistical language models in mining natural source code naming conventions. They extended their work to suggest renaming methods based on their bodies and renaming classes based on its own methods. Their recommendation uses a neural probabilistic language model to input a set of words, coded in a vector, which is fed to a hidden layer of the neural network. The output layer evaluates the conditional probability of each word in the vocabulary given the input sequence.

Liu et al.[23] proposed an approach that monitors the rename activities performed by developers and then recommends a batch of rename operations to all closely related code elements whose names are similar to that of the renamed element by the developer. Then, they extended their approach to make it independent from conducted renaming operations [24].

4 EXPERIMENTAL SETUP

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 1 depicts the flow of steps involved in this experiment. Described below are details of each phase.

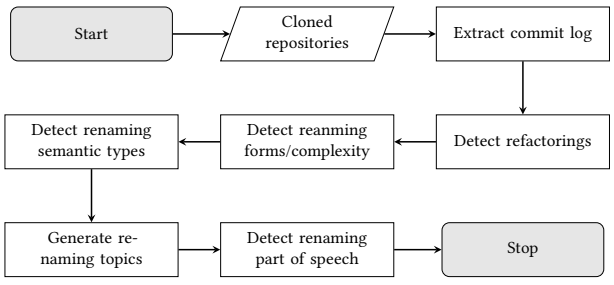

Figure 1: Overview of Experiment Methodology

4.1 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 [5]. To identify refactoring operations performed by developers on these projects, we use RefactoringMiner [30] on each project. By 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 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.

Table 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%

4.2 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 [8]. To this end, we utilize the Natural Language Toolkit (NLTK, https://www.nltk.org/), which has an implementation of Wordnet [26], 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 [20]. Table 2 provides an overview of the most frequent number of terms that constitute the name of the identifier for packages, classes, and methods.

Table 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%***

Our semantic analysis follows the approach presented in [8]. 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 recompare 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 [8] to determine semantic changes as summarized in Section 2 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.

5 EVALUATION

This section reports on the findings we observed when analyzing the rename refactorings.

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

As shown in Table 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 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.

Table 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%***	

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

Table 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 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.

Table 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%***	

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

As depicted in Figure 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.

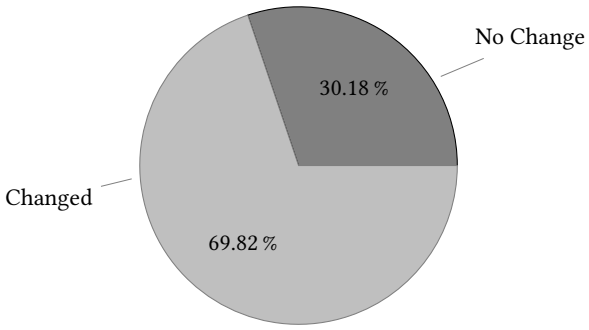

Figure 2: Part of Speech Changes for 42,060 Matched Terms

The most common grammar changes are depicted in Table 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 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 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 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).

Table 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%***

Table 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%***

5.4 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) [11] to discover and generate topics contained within the commit messages automatically. To this end, we utilize the LDA implementation contained within gensim [28], 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 7 for each of the seven types of semantic change classifications described in this paper 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 7 only presents the top 2 topics (labeled A and B) for each category due to space constraints. 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 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), 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.

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

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

In RQ1, using Table 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 2). Additionally, from RQ2, using Table 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 8, which shows how many simple and complex renamings are categorized 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.

Table 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%***	

6 THREATS TO VALIDITY

The focus of this study is limited to open source Java projects and so may not be representative. However, we mitigate this threat by performing the study on a large and diverse (in size, age, and committers) corpus of systems. The type and accuracy of detected renaming refactorings are limited to RefactoringMiner. However, RefactoringMiner is frequently used in research, has high precision/recall scores, and the analysis of a large quantity of method, class, and package identifiers are beneficial. Additionally, our study on the semantic characteristics of renamed identifiers is limited to the WordNet corpus and algorithms in NLTK. Other tools for analyzing identifiers may give varying results. We tried to obtain REPENT for use in this study; however, it was unavailable. Instead, we use a toolset very close to what was used in REPENT [8].

Finally, our results did differ from the original study [8] in one way that is significant to this study: The distribution of semantic changes. For example, we found very few change meaning renames, but Arnaoudova et al. found that they were one of the most common. We found many narrow meaning renames, but Arnaoudova found few. We did not directly compare to their work because of the fact that Arnaoudova considered all identifiers while we only considered method, class, and package identifiers. They also did not report numbers for identifiers separated by where they were found (e.g., in a class name). Combined with the fact that we could not use REPENT, we felt it was inappropriate to directly compare results.

7 CONCLUSIONS AND FUTURE WORK

We present a study that analyzes the identifier names of 500k rename refactorings on Java methods, classes, and packages; fitting each rename to this taxonomy and then contextualizing these renames by analyzing commit messages. We believe that a firm grasp of what these renames manifest is important because techniques that recommend, appraise, or generate identifiers are trying to mimic any developer-centric activity in order to predict what a developer would accept. By investigating what causes a developer to, for example, perform complex renamings that broaden the meaning of an identifier versus simple renamings that narrow an identifier's meaning, we can more accurately model how developers think about identifier names and their evolution. Further, we can more intelligently suggest words that adequately fit in the type of changes and maintenance tasks developers are assigned to.

In our future study, we plan to extend this work to include more types of identifiers. We will also begin using static analysis techniques to analyze source code changes that influence different types of renamings. We would like to see how changes to the code surrounding an identifier influence the renaming action taken by developers and will use some of the results from this study to help direct the process. This may mean looking at refactorings and other changes that occur in tandem with rename refactorings.

REFERENCES

[1] *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[2] 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.

[3] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 281–293. ACM, 2014.

[4] M. Allamanis, E. T. Barr, C. Bird, and C. 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.

[5] M. Allamanis and C. 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.

[6] 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.

[7] V. Arnaoudova, M. Di Penta, and G. Antoniol. Linguistic antipatterns: What they are and how developers perceive them. *Empirical Software Engineering*, 21(1):104–158, 2016.

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

[9] V. Arnaoudova, M. D. Penta, G. Antoniol, and Y. G. GuÃľhÃľneuc. A new family of software anti-patterns: Linguistic anti-patterns. In *2013 17th European Conference on Software Maintenance and Reengineering*, pages 187–196, March 2013.

[10] D. Binkley, M. Hearn, and D. Lawrie. Improving identifier informativeness using part of speech information. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 203–206, New York, NY, USA, 2011. ACM.

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

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

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

[14] F. Deissenbock and M. Pizka. Concise and consistent naming [software system identifier naming]. In *13th International Workshop on Program Comprehension (IWPC'05)*, pages 97–106, May 2005.

[15] L. Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, May 2000.

[16] 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.

[17] E. Hill. *Integrating Natural Language and Program Structure Information to Improve Software Search and Exploration.* PhD thesis, Newark, DE, USA, 2010. AAI3423409.

[18] E. Hill, Z. P. Fry, H. Boyd, G. Sridhara, Y. Novikova, L. 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.

[19] E. W. Host and B. M. Ostvold. Debugging method names. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 294–317, Berlin, Heidelberg, 2009. Springer-Verlag.

[20] M. Hucka. Spiral: splitters for identifiers in source code files. 2018.

[21] 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.

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

[23] 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.

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

[25] N. Meng, M. Kim, and K. 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.

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

[27] 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.

[28] R. Řehůřek and P. 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.

[29] 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.

[30] D. Silva, N. Tsantalis, and M. T. 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, pages 858–870, New York, NY, USA, 2016. ACM.

[31] G. Sridhara, E. Hill, D. Muppaneni, L. 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.

[32] B. Zhang, E. Hill, and J. Clause. Towards automatically generating descriptive names for unit tests. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 625–636, Sept 2016.