Contextualizing Rename Decisions using Refactorings and Commit Messages

Anthony Peruma*, Mohamed Wiem Mkaouer*, Michael J. Decker†, Christian D. Newman*
*Rochester Institute of Technology, Rochester, NY, USA
†Bowling Green State University, Bowling Green, OH, USA
axp6201@rit.edu, mwmvse@rit.edu, mdecke@bgsu.edu, cnewman@se.rit.edu

***Abstract*—Identifier names are the atoms of comprehension; weak identifier names decrease productivity by increasing the chance that developers make mistakes and increasing the time taken to understand chunks of code. Therefore, it is vital to support developers in naming, and renaming, identifiers. In this paper, we study how terms in an identifier change during the application of rename refactorings and contextualize these changes using co-occurring refactorings and commit messages. The goal of this work is to understand how different development activities affect the type of changes applied to names during a rename. Results of this study can help researchers understand more about developers' naming habits and support developers in determining when to rename and what words to use.**

***Index Terms*—Program Comprehension, Identifier Names, Rename Refactoring**

I. Introduction

Program comprehension is an extremely important part of a developer's day-to-day activities. Before a developer can perform any activity related to maintaining an existing code base, they must first read and understand the code corresponding to this activity. In short, comprehension is critical to all software maintenance and evolution activities.

Previous work has shown that developers spend a significant amount of time comprehending code [1], [2]. Some have estimated that developers spend ten times longer reading and analyzing code as opposed to writing it [2]. In order to help developers perform optimally when maintaining large code bases, it is critical that we reduce the amount of time they spend understanding code by making it easier for them to quickly and effectively understand the code they will be working on [3]–[6].

One important aspect of comprehension is the choice of naming identifiers (e.g., class names, method names, etc.). Identifier names are the atoms of comprehension; weak identifier names decrease productivity [4]–[7], normalizing identifier names helps both developers and research tools [8], [9], and many research projects, both recent and otherwise, have an explicit goal of improving identifier naming in source code using algorithms that take advantage of large datasets of identifier names and/or static analysis [10]–[15].

One way to improve identifiers is to apply a rename refactoring [16]. The rename refactoring is defined as modifying the name of an identifier without modifying the intended behavior of the code of which the identifier is part. Many Integrated Developer Environments (IDE) offer a built-in rename refactoring functionality. Unfortunately, even though there is some support for the mechanical act of renaming via IDEs, there is little support to help inform developers of when to rename (i.e., 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 which 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.

Because naming heavily effects comprehension, it is important to fully support developers when they must modify identifier names. That is, research must support developers in applying rename refactorings. Current research on naming focuses heavily on suggesting identifier names [10], [11], [13], [14], studying how names correlate with behavior [12], [17], and analyzing names [18]–[22]. However, there are few studies that investigate how names evolve [18], [23]–[25] (i.e., are changed via rename) and how these changes correlate with changes made to source code (i.e., whether behavior-preserving or not) and as part of a larger development plan.

To help fill the gap in knowledge concerning renaming identifiers, this paper studies rename refactorings in two ways. 1) This paper utilizes a taxonomy of rename types published by Arnoudova et al. [18] 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 paper contextualizes these rename types by analyzing commit log data and changes made directly to the source code. This allows us to understand how changes to the code surrounding an identifier affects the identifier's name and, likewise, how development activities (i.e., written in a commit log) affect the identifier's name. This work extends the research started by Peruma et al. [23]. We first enhance the topic modeling analysis through the use of topic coherence and n-grams. Next, we extend the contextualizing activity to include code refactorings that surround renames.

The goal of this study is to forward our understanding of the changes made to identifiers during renaming activities by studying the two perspectives described in the previous paragraph. In the long term, the outcomes of this study will

2470-6892/19/$31.00 ©2019 IEEE
DOI 10.1109/SCAM.2019.00017

be used to 1) recommend when a rename should be applied, 2) recommend the types of words to use when applying a rename, and 3) develop a model that describes how developers mentally synergize names using domain and project knowledge. Additionally, we reflect on challenges for future research in analyzing data similar to what was used in this paper.

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. This will inform us of the types of developers in our rename data.

RQ2: What are the refactorings that occur more frequently with identifier renames? With this question, we aim to understand what types of refactorings tend to occur before or after a rename. Our theory 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.

II. RELATED WORK

Since the choice of adequate naming for identifiers is critical for code understandability, there have been many studies that have analyzed the quality of identifiers and how identifier quality affects comprehension and developer efficiency.

Arnoudova et al. [18] proposed an approach to analyze and classify identifier renamings. They mined several rename 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 and conducted a survey showing that 68% of developers think recommending identifier names would be useful. The same researchers also defined a catalog of linguistic anti-patterns that are found to deteriorate the quality of code understanding [17]. They have shown 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.

Liu et al. [24] 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. They also studied the relationship between argument and parameter names and use the patterns they found to detect naming anomalies and suggest renames to developers [25]. Peruma et al. [23] studied how terms in an identifier change and contextualized these changes by analyzing commit messages using a topic modeler; looking for words that indicate what development activity had occurred with the change. We extend this work by examining refactoring co-occurrence with renames and analyzing commit messages of these co-occuring refactorings.

Høst and Østvold [12] designed automated naming rules using method signature elements, i.e., return type, parameters 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. Binkley et al. [21] presented empirically-derived rules that certain types of identifiers (e.g., class field identifiers) should follow. One of these rules is that class fields should never be just an adjective.

There are several recent approaches to appraising identifier names for variables, methods, and classes. Kashiwabara et al. [14] 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. [13] uses 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. Allamanis et al. [10] use a novel language model called the Subtoken Context Model, which is a neural network that has some similarity to n-grams (in that it uses a previously seen set of tokens to predict a new token). The difference is that the neural network is able to take into account long-distance features (e.g., identifier names that occur very far away from the target location) and produce neologisms (essentially, new identifiers) by concatenating words together (i.e., as is commonly done by developers).

Liblit et al. [27] discusses naming in several programming languages and makes observations about how natural language influences the use of words in these languages. Schankin et al. [4] 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 [5] compared comprehension of identifiers containing words against identifiers containing letters and/or abbreviations. Their results show that when identifiers contained only words instead of abbreviations or letters, developer comprehension speed increased by 19% on average. Lawrie et al [6] did a study on 100+ programmers; asking them to describe twelve different functions. These functions used 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’s work [7] extends their previous work on java class identifiers [28] to show that flawed method identifiers are also (i.e., along with class identifiers) associated with low-quality code according to static analysis-based metrics.

III. ANALYSIS OF RENAMES

The hypothesis of this paper is as follows: Changes to the name of an identifier are most likely related to other changes made locally (i.e., in the same class, function, or file) and the intuition behind those changes. Under this hypothesis, we

should be able to correlate the types of changes made to a name with other local changes. In this section we will take a look at a couple examples of this occurrence found in our data set. We use a taxonomy originally created by Arnaoudova et al [18] and used in a study similar to this one by Peruma et al [23] 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.

A. 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 reflects the lexical change made to the identifier. It is broken 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 changes 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 the identifier's **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 the identifier's **meaning 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., 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 broadened) 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*– 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).

B. 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 and refactoring operations, is reflected in their renaming 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.

Finally, commit messages are not the only way to contextualize rename refactorings. Changes to the code surrounding a name also help in understanding what 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 [16]. 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 [29] 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" [30].

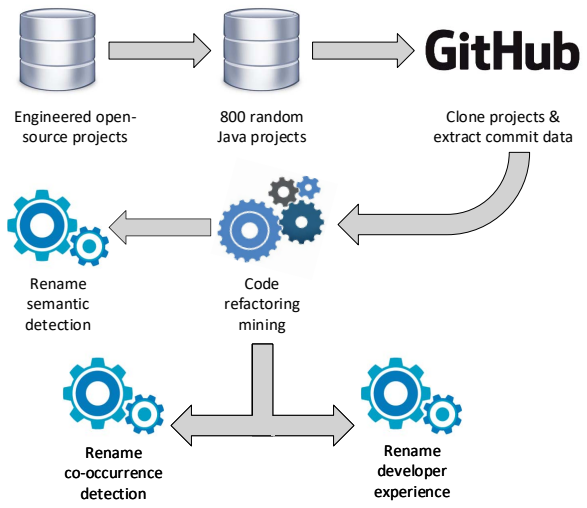

Fig. 1: Methodology overview

Another example comes from a move class refactoring, where a class was moved from one package to another [31]. This refactoring commit had the following comment: "Incremental changes, some package refactorings etc". Further, a rename was performed after this commit: *JsonViewResult→JsonView* [32]. 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. The question we ask, in the context of these examples, is whether there are overarching themes to the way names change given that a refactoring has occurred in a commit surrounding it. If so, then it is possible to study these trends and use them to support developers in their naming activities.

IV. Methodology

Our experiment methodology consisted of two phases - Data Collection and Detection. The Data Collection phase consisted of building our dataset while the Detection phase consisted of analyzing and querying the dataset for specific features to help answer our research questions. Depicted in Figure 1 is an overview of the approach that we undertook to conduct our experiments. In the subsequent subsections, we explain in detail the approach for each activity.

A. Data Collection Phase

Projects: To obtain a viable dataset to perform our experiments on we selected 800 random, curated open source Java projects hosted on GitHub. These curated projects were selected from a dataset made available by [33]. The authors of this dataset classified engineered software projects based on the projects use of software engineering practices such as documentation, testing, and project management. 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. In total, we collected 748,001 commits with a project containing 732 commits and 19 developers on average.

TABLE I: 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%*

Refactorings: Our study utilizes RefactoringMiner [34] to mine refactorings occurring in the source code. At the time of conducting our study, Refactoring Miner can identify 28 different refactoring operations. From this list of operations, seven are rename based operations. RefactoringMiner iterates over all commits of a repository in chronological order and compares the changes made to Java source code files by developers and detects refactorings in the code based on a predefined set of refactoring rules. We investigated 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, our experiments were conducted on the entire commit history of the project (and not on a release-by-release comparison). In total, we detected 711,495 refactoring operations with each project in our dataset exhibiting refactoring operations. 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. Due to space constraints, we present only the top five refactoring operations, that was mined from our dataset, in Table I. The entire list is available on our website [35].

B. Detection Phase

Rename Forms & Semantics: To detect the form and semantic update an identifier undergoes we obtained the program utilized in [23] and made some minor updates to the semantic categorization logic. The program primarily relies on Python's Natural Language Toolkit (NLTK) [36] to compare the original and renamed identifier name to determine the type of semantic change made by the developer. Table II shows the distribution of rename form and semantic meaning types in our dataset.

Rename Co-occurrence: We built a custom program to detect refactorings that occur before and after rename refactoring by iterating over the mined refactoring-based commits in our dataset. Since our rename refactorings are related to classes, attributes, methods, method parameters, and method variables, we restricted our occurrence detection to refactorings that are applied to only these types of elements. For each renamed element type, we first extract all unique instances. Next, we iterated through all refactorings searching for refactorings that involved the specific instance.

TABLE II: Distribution of rename forms and semantic meanings

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%

To better highlight this process, consider the example where we detected the class `stormpot.CountingAllocatorWrapper` as being renamed to `stormpot.CountingAllocator` [37]. We then queried our list of unique attributes, methods, parameters, and variables for elements that were part of this class and had also undergone a refactoring. Our search resulted in an attribute, `counter`, belonging to this class and had undergone a rename refactoring (prior to the class being renamed) [38]. Finally, we record this pair of refactorings in our database.

Commit Log Contextualizing: As part of our experiment, we aimed to contextualize certain instances of the commit log in order to derive the developer's rationale for performing specific tasks. To achieve this, we performed a topic modeling and n-gram analysis of commit messages. We utilized the latent Dirichlet allocation (LDA) [39] algorithm for our topic modeling analysis. We used a combination of topic coherence [40] and manual empirical analysis to determine the ideal number of topics, as past research has shown that the number of topics can vary between studies and datasets [41]. A prerequisite to these activities was a text preprocessing task where we cleaned and standardized the commit messages. Some key steps in our preprocessing included: removal of stopwords, URLs, numeric and alphanumeric characters/words, and non-dictionary words. Additionally, we also expanded contractions and performed stemming and lemmatizing on words.

Developer Experience: Since obtaining the experience of a developer can be subjective and also not entirely feasible for a large scale study such as this, we conducted a more objective-based experiment. To this extent, we followed the approach utilized by [42]. 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 Developers Commit Ratio (DCR). In this ratio, we measure the number of individual commits made by the developer against all project commits. In other words, DCR = $(\frac{IndividualContributorCommits}{TotalAppCommits})$. To mitigate the threat of mis-attributing commits due to the use of git features such as pull requests, we only consider the author of a commit as its developer.

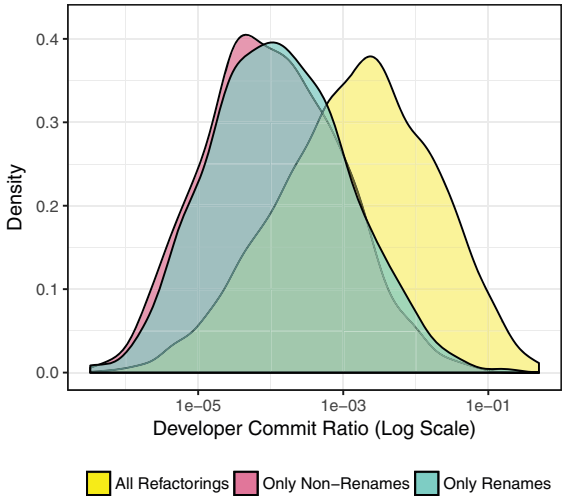

Fig. 2: Distribution of DCR values for developers based on the type of refactoring performed in their project

V. Experimental Results

We will now discuss our results. The discussion is broken down into our three Research Questions. In RQ1, we discuss the experience of developers that perform rename refactorings versus other types of refactorings. In RQ2, we look at what kinds of refactorings happen before or after a rename refactoring and discuss both how often rename refactorings are preceded or followed by another refactoring and what types of refactorings these preceding or following changes represent. In RQ3, we combine and discuss data from RQ2 with commit message information and the semantic change types discussed in Section III-A with a goal of using the commit message and refactoring information to contextualize the semantic change types we detected in our set of renames.

A. 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 followed the same approach as [42]. Since the number of developers in each project differs, we calculated an adjusted DCR value for each developer by dividing the developers original DCR value by the number of developers in the project. We also restricted our experiment to projects that had only two or more developers.

As shown in Figure 2, it is not surprising that developers who perform all types of refactorings have a higher DCR than those that perform only rename refactorings. However, it is interesting to observe 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 performed 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)

TABLE III: Distribution of rename form and semantic meaning updates performed by developers based on their refactoring preferences

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%

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 the renaming is a difficult problem [18].

Looking at the different types of identifier rename forms, we observed that there was 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 III provides a breakdown on the distribution of rename form and semantic meaning updates.

Our experiment on developer experience shows the developers with more project experience (i.e., contributions) are more accustomed to performing a multitude of different types of refactoring operations. This should not be 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 renames have wide 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). However, there is no difference between these two types of user groups with regards to the complexity (i.e., form and semantic) of the rename. Because developers who apply renames may have limited project experience, we must keep this factor in mind when analyzing changes made to names.

B. 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 studied *the type of refactoring commits that occur just before and after a rename refactoring commit*. This part of our study focused on the renames of classes, attributes, methods, method parameters, and method variables. For each entity type, we extracted the list of unique instances that underwent a rename and then searched for the refactoring that directly preceded and directly followed (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).

TABLE IV: 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*

Interestingly, we observed that for all elements that are subject to renames, developers frequently perform the rename in isolation. In other words, approximately 90% of the rename refactorings did not have a refactoring occurring immediately before and after the rename. However, this does not mean that the developer only performed a rename to the element during the lifetime of the element. There can be other non-refactoring activities that were applied to the element by the developer that is not considered a refactoring (e.g., adding lines of code to a method). For scenarios where there are refactorings either before or after a rename, we noticed that more operations occur before a rename ($\approx 8\%$) than after ($\approx 2\%$).

In general, the majority of the refactorings that occur before a rename are related to changes/updates to functionality. Additionally, we also observed that some of these commits are also bug fix related and also 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 [43]. Thereafter the developer renames the newly created method to `getClassUrl` to ensure that name follows "Google's style rules" [44].

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 [45]. However, in a subsequent commit [46], the developer removes this functionality from the method and also reverts back to the original method name.

As the majority or 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 IV 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. The complete list of refactorings that proceed and follow a rename refactoring is available on our project website.

C. Class Rename

Our study of class renames involved 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 observed 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 [47] 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 [48] 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 observed that the majority of renames ($\approx 7.15\%$) occur in the commit immediately following the move. It is also interesting to note that a gap of between 1 to 5 commits occurs around 27.73% of the time between a class move and rename.

D. 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 [49], the developer moves the attribute `jobId` with the message "added the jobId to a few more logs". The subsequent refactoring commit [50] for this attribute involves a renaming operation in which the attribute is renamed to `context` as part of a "cleanup" activity.

We observed 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.

E. Method Rename

For methods, we investigated 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 observed 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 observed that the reason for the initial rename is due to developers changing the behavior/purpose of the method. Furthermore, we noticed that the second occurrence of the method rename reverts the first rename operation. For example, in [51], 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 [52], 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.

F. Method Variable Rename

Like methods, method variables also undergo rename operations in succession. Once again, looking at the commit messages, we can gauge that the reason for the initial rename was 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 [53] the developer renames the variable `drop` to `assembledDrop` with the message "simplified drop assembly a bit". The next commit reverts the variable name when the developer performs a "misc code cleanup" activity.

Summary for RQ2: We have shown 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. Of this subset, we observed that a majority of the time developers perform a refactoring operation just before the rename, these two operations happen in a short (commit) interval. Finally, in situations where a rename follows another rename, it has been observed that developers revert to the original name when performing the second rename.

TABLE V: An overview of the types of semantic updates an identifier name undergoes after a refactoring

Element Type	Refactoring Before Rename	Type of Semantic Update	Top 3 Semantic Change Subtypes
Class	Move Class	Change (#: 2,659; %: 84.14%) Preserve (#: 501; %: 15.85%)	Narrow (63.56%) Broaden (28.13%) Add (3.65%)
	Rename Method	Change (#: 1,961; %: 90.0%) Preserve (#: 218; %: 10.0%)	Narrow (57.42%) Broaden (31.56%) Add (6.78%)
	Rename Variable	Change (#: 1,479; %: 100.0%)	Narrow (100%)
Attribute	Move Attribute	Change (#: 1,419;%: 94.66%) Preserve (#: 80;%: 5.34%)	Add (54.05%) Narrow (24.59%) Broaden (16.07%)
	Pull Up Attribute	Change (#: 187;%: 85.0%) Preserve (#: 33;%: 15.0%)	Narrow (66.84%) Broaden (25.67%) Add (3.21%)
	Push Down Attribute	Change (#: 47;%: 63.51%) Preserve (#: 27;%: 36.49%)	Narrow (70.21%) Broaden (23.4%) Add (2.13%)
Method	Rename Method	Change (#: 1,752;%: 81.19%) Preserve (#: 406;%: 18.81%)	Narrow (36.42%) Broaden (31.16%) Add (24.14%)
	Extract Method	Change (#: 1,447;%: 85.42%) Preserve (#: 247;%: 14.58%)	Narrow (64.06%) Broaden (26.12%) Add (4.49%)
	Rename Variable	Change (#: 840;%: 87.41%) Preserve (#: 121;%: 12.59%)	Narrow (49.28%) Broaden (32.86%) Remove (9.17%)
Variable	Rename Variable	Change (#: 3,033;%: 98.89%) Preserve (#: 34;%: 1.11%)	Add(77.35%) Narrow (14.93%) Broaden (6.17%)
	Extract Variable	Change (#: 281;%: 92.13%) Preserve (#: 24;%: 7.87%)	Narrow (71.17%) Broaden (19.57%) Add(6.05%)
	Inline Variable	Change (#: 6; %: 100.0%)	Add (66.67%) Narrow(33.33%)

G. 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 looked at the types of semantic changes applied to identifier names given that another refactoring was applied in the previous commit. We then analyzed this data to understand whether the refactoring that happened before the rename had any affect on the semantic change applied during the rename. Additionally, we performed commit message analysis 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. Please refer to III-A for a refresher on the semantic change categories. Table V highlights the distribution of these change types for elements that undergo a rename after another type of refactoring operation.

We observed 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 [54], 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" [55]. 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 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 [56], 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" [57]. Finally, adding to the identifier name was the third most frequent type of semantic change.

There are a few interesting things to point out from Table V. The first is that a Rename Variable followed by another Rename Variable tended to add meaning instead of narrow or broaden. The same applies for renames occurring after a Move Attribute refactoring and after a 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 used LDA and n-gram analysis on commit messages associated with the rename refactorings occurring after a refactoring operation. A previous work has also used LDA on a similar context [23], 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. Technically, we extended [23] topic modeling approach by incorporating additional text preprocessing and the use of topic

TABLE VI: 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 VII: 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)

coherence scores in order to improve the quality of our text analysis compared to the original paper. The results of this analysis are in Tables VI, VII VIII, and IX. 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 VI 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. We analyzed the commit messages associated with these topics and found the updates are due to bug fixes or code optimizations. For example in commit [58], the broadening of the name is associated with the message "...Made the factory generic",

TABLE VIII: 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 IX: 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)

which a broaden meaning rename would logically follow. Table VII 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 [59], 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 VIII 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 [60] and legacy [61] code is a task associated with code cleanup activities. These activities can also range from simple identifier renames [62] to more intensive structural changes [63]. Finally, Table IX 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 revealed 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 [64], 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 included '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 original expected behavior of the identifier. For example, in [65] as part of updates to the user interface, the developer performs a *Pull Up Method* operation on the method `calcTotal`. The next update [66] 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. A narrowing (i.e., specialization) of the name is the most common type of change in meaning. 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, but further research using additional artifacts, and new methods of natural language text analysis for software engineering, are required to provide us with stronger insights.

VI. Threats to Validity

Our experiments are based only on well-engineered Java systems [33], meaning the results may not generalize to systems written in other languages. The type and volume of detected renaming refactorings are limited to RefactoringMiner's capabilities. However, RefactoringMiner is currently the most accurate refactoring detection tool [67] and is widely used in research concerning refactorings.

Our experiment on developer experience utilized project contributions as a proxy for the developer's experience; an approach we followed from a similar study. As with many software metrics, this metric is not perfect, and may not always appropriately measure experience.

Part of this work analyzed commit messages. To mitigate bias in deciding the terms to present after commit message analysis, we used a peer-review approach. The authors reviewed the entire list of generated terms and decisions that were made had to be unanimous. Furthermore, we manually referred to the entire commit message to verify the context around the terms of interest.

Because of our partial reliance on NLTK to detect semantic changes performed via rename, there is a threat that some of the conclusions drawn by the semantic change detection algorithm may be inaccurate. We mitigate this by thorough testing of the tool, but it is known that tools trained specifically on software engineering data tend to generalize better than tools trained on general natural language data and applied to source code [8], [68]. Unfortunately, there are no models for software data that offer word relation data similar to NLTK.

VII. Conclusion, Discussion, Future Work

In this paper we used refactorings, static analysis, and commit messages to understand characteristics of changes applied to names and to determine if these changes correlate to different developer activities (e.g., narrowing of a name after applying extract method). Our long term goal is to support recommendation of when/how to rename identifiers and to understand more about developer naming mental models. This study brings us a step closer to achieving this goal by showing us some interesting trends in developer behavior with respect to renaming as well as by highlighting where our methodology is weak or where we lack data. We discuss our findings below.

RQ1 shows us that developers with relatively less experience than their peers have a higher likelihood of applying rename refactorings than other types of refactorings. RQ2 shows us what types of refactorings happen before a rename and that there are some specific terms that are associated with these refactorings which can help us understand some of the motivation behind renames occurring directly afterward and help us determine when we should suggest applying a rename. Other work has similarly shown how specialized terminology indicates developer refactoring activities [69]. The data also indicates that the vast majority of renames are not correlated with any refactoring occurring before or after their application.

More research into these non-refactoring changes is required since only a minority of renames happen directly after a refactoring. Finally, in RQ3, we saw that there are generally three types of semantic changes that frequently occur during a rename applied after another refactoring: Narrow, Broaden, and Add meaning. When analyzing commit messages associated with each of these, we were able to identify terms which indicate development tasks associated with the type of rename. While somewhat indicative of the larger context, these terms were too isolated and required us to manually analyze commit messages for more context. On the positive side, our data does show that the motivation for semantic changes is recorded and can be detected; allowing us to understand more about how names evolve in the larger software evolution context.

However, it also shows that a significant amount of work is needed to automatically derive these motivations more effectively from commit messages, other natural language software artifacts, and general source code changes. In particular, the biggest problems we faced with analyzing large numbers of commit messages is that: 1) the terms frequent enough to be detected are high-level and not descriptive of individual project efforts (e.g., we can determine that projects are performing structure changes, but not what types of structural changes or why). Also, 2) the commit messages often simply do not contain enough information, potentially indicating the need for more natural language software artifacts which will likely be more challenging to analyze automatically. An effective method for performing this type of analysis would positively impact our ability to support developers in assigning names, renaming, and suggesting when and where names need to evolve. The work we present in this paper shows that this context is obtainable, but there are still many challenges to it.

In future work, we plan to investigate more effective means of analyzing commit messages and other natural language software artifacts to help us address the problems discussed above. Additionally, we intend to investigate the use of software differencing techniques [70], [71] to allow us to analyze general software changes that occur around a rename. Finally, the dataset utilized in this study is available on our project website [35].

VIII. Acknowledgements

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

REFERENCES

[1] T. A. Corbi, "Program understanding: Challenge for the 1990s," *IBM Systems Journal*, vol. 28, no. 2, pp. 294–306, 1989.

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

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

[4] A. Schankin, A. Berger, D. V. Holt, J. C. Hofmeister, T. Riedel, and M. Beigl, "Descriptive compound identifier names improve source code comprehension," in *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18, (New York, NY, USA), pp. 31–40, ACM, 2018.

[5] 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)*, pp. 217–227, Feb 2017.

[6] 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)*, pp. 3–12, June 2006.

[7] 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*, pp. 156–165, IEEE, 2010.

[8] D. Binkley, D. Lawrie, and C. Morrell, "The need for software specific natural language techniques," *Empirical Softw. Engg.*, vol. 23, pp. 2398–2425, Aug. 2018.

[9] 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 *Proceedings of the 35th IEEE International Conference on Software Maintenance*, IEEE, 2019.

[10] 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, (New York, NY, USA), pp. 38–49, ACM, 2015.

[11] K. Liu, D. Kim, T. F. Bissyand, T. Kim, K. Kim, A. Koyuncu, S. Kim, and Y. 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), ACM, 2019.

[12] E. W. Høst and B. M. Østvold, "Debugging method names," in *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, (Berlin, Heidelberg), pp. 294–317, Springer-Verlag, 2009.

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

[14] 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)*, pp. 323–327, Feb 2014.

[15] C. D. Newman, A. Peruma, and R. AlSuhaibani, "Modeling the relationship between identifier name and behavior," in *Proceedings of the 35th IEEE International Conference on Software Maintenance*, IEEE, 2019.

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

[17] V. Arnaoudova, M. Di Penta, G. Antoniol, and Y. Guhneuc, "A new family of software anti-patterns: Linguistic anti-patterns," in *2013 17th European Conference on Software Maintenance and Reengineering*, pp. 187–196, March 2013.

[18] V. Arnaoudova, L. M. Eshkevari, M. D. Penta, R. Oliveto, G. Antoniol, and Y.-G. Gueheneuc, "Repent: Analyzing the nature of identifier renamings," *IEEE Trans. Softw. Eng.*, vol. 40, pp. 502–532, May 2014.

[19] 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)*, pp. 228–239, Feb 2017.

[20] 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)*, pp. 1–6, Sep. 2015.

[21] 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, (New York, NY, USA), pp. 203–206, ACM, 2011.

[22] 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)*, pp. 3–12, May 2013.

[23] A. Peruma, M. W. Mkaouer, M. J. Decker, and C. D. Newman, "An empirical investigation of how and why developers rename identifiers," in *International Workshop on Refactoring 2018*, 2018.

[24] H. Liu, Q. Liu, Y. Liu, and Z. Wang, "Identifying renaming opportunities by expanding conducted rename refactorings," *IEEE Transactions on Software Engineering*, vol. 41, no. 9, pp. 887–900, 2015.

[25] 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*, pp. 1063–1073, IEEE, 2016.

[26] G. A. Miller, "Wordnet: a lexical database for english," *Communications of the ACM*, vol. 38, no. 11, pp. 39–41, 1995.

[27] 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.

[28] 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*, pp. 31–35, Oct 2009.

[29] https://github.com/coremedia/jangaroo-tools/commit/7a494f1.

[30] https://github.com/coremedia/jangaroo-tools/commit/fc54b3f.

[31] https://github.com/3wks/thundr/commit/53aaf15.

[32] https://github.com/3wks/thundr/commit/9b02920.

[33] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating github for engineered software projects," *Empirical Software Engineering*, vol. 22, pp. 3219–3253, Dec 2017.

[34] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, (New York, NY, USA), pp. 483–494, ACM, 2018.

[35] "Project website." https://sites.google.com/g.rit.edu/scanl/.

[36] S. Bird, E. Klein, and E. Loper, *Natural language processing with Python: analyzing text with the natural language toolkit*. "O'Reilly Media, Inc.", 2009.

[37] https://github.com/chrisvest/stormpot/commit/459d423.

[38] https://github.com/chrisvest/stormpot/commit/d2931d3.

[39] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *Journal of machine Learning research*, vol. 3, no. Jan, pp. 993–1022, 2003.

[40] M. Röder, A. Both, and A. Hinneburg, "Exploring the space of topic coherence measures," in *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining*, WSDM '15, (New York, NY, USA), pp. 399–408, ACM, 2015.

[41] A. Barua, S. W. Thomas, and A. E. Hassan, "What are developers talking about? an analysis of topics and trends in stack overflow," *Empirical Software Engineering*, vol. 19, pp. 619–654, Jun 2014.

[42] 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)*, pp. 165–169, May 2017.

[43] https://github.com/stripe/stripe-java/commit/4fdadaf.

[44] https://github.com/stripe/stripe-java/commit/19d4d5a.

[45] https://github.com/atlasapi/atlas-model/commit/4da9fc2.

[46] https://github.com/atlasapi/atlas-model/commit/fc19c98.

[47] https://github.com/heroku/heroku.jar/commit/008dbc2.

[48] https://github.com/heroku/heroku.jar/commit/0c1c18d.

[49] https://github.com/mapfish/mapfish-print/commit/bc1f422.

[50] https://github.com/mapfish/mapfish-print/commit/fe44bd1.

[51] https://github.com/davemckain/qtiworks/commit/2a1f9df.

[52] https://github.com/davemckain/qtiworks/commit/9cb51b2.

[53] https://github.com/mung3r/ecocreature/commit/42e5d9f.

[54] https://github.com/ismavatar/lateralgm/commit/2d1bdaf.

[55] https://github.com/ismavatar/lateralgm/commit/e41c4c5.

[56] https://github.com/liveramp/jack/commit/762b540.

[57] https://github.com/liveramp/jack/commit/b331247.

[58] https://github.com/motech/ananya-kilkari/commit/b3b95f4.

[59] https://github.com/buchen/portfolio/commit/1bdeccb.
[60] https://github.com/eclipse-vertx/vert.x/commit/921c69e.
[61] https://github.com/jetbrains/teamcity-nuget-support/commit/da10d2c.
[62] https://github.com/davemckain/qtiworks/commit/0c924ab.
[63] https://github.com/jrebirth/jrebirth/commit/d82fb1b.
[64] https://github.com/unquietcode/flapi/commit/4586325.
[65] https://github.com/buchen/portfolio/commit/9fc2fad.
[66] https://github.com/buchen/portfolio/commit/e1d7472.
[67] L. Tan and C. Bockisch, "A survey of refactoring detection tools," in *Software Engineering*, 2019.
[68] R. Jongeling, P. Sarkar, S. Datta, and A. Serebrenik, "On negative results when using sentiment analysis tools for software engineering research," *Empirical Software Engineering*, 01 2017.
[69] E. A. Alomar, M. W. Mkaouer, and A. Ouni, "Can refactoring be self-affirmed? an exploratory study on how developers document their refactoring activities in commit changes," in *Proceedings of the 3rd International Workshop on Refactoring*, (New York, NY, USA), ACM, 2019.
[70] M. J. Decker, *srcDiff: Syntactic Differencing to Support Software Maintenance and Evolution*. PhD thesis, 2017.
[71] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, (New York, NY, USA), pp. 313–324, ACM, 2014.