

Using Grammar Patterns to Interpret Test Method Name Evolution

Anthony Peruma*, Emily Hu[†], Jiajun Chen[‡], Eman Abdullah AlOmar*,
Mohamed Wiem Mkaouer*, Christian D. Newman*

*Rochester Institute of Technology, Rochester, NY, USA

[†]Tufts University, Medford, MA, USA

[‡]Stony Brook University, Stony Brook, NY, USA

axp6201@rit.edu, emily.hu@tufts.edu, jiajun.chen.2@stonybrook.edu, eman.alomar@mail.rit.edu,
mwmvse@rit.edu, cnewman@se.rit.edu

Abstract—It is good practice to name test methods such that they are comprehensible to developers; they must be written in such a way that their purpose and functionality are clear to those who will maintain them. Unfortunately, there is little automated support for writing or maintaining the names of test methods. This can lead to inconsistent and low-quality test names and increase the maintenance cost of supporting these methods. Due to this risk, it is essential to help developers in maintaining their test method names over time. In this paper, we use grammar patterns, and how they relate to test method behavior, to understand test naming practices. This data will be used to support an automated tool for maintaining test names.

I. INTRODUCTION

In software, test methods names are constructed to describe both the entity that is being tested as well as actions taken by the test [1]. The name of a test is important for the same reason production method names are important; they help developers understand the purpose of the method. Further, these names can be used by automated approaches to analyze/understand test methods and automatically generate code for the test methods. Prior research indicates that test method names have a different structure than production method names [1], [2], but understanding how they are similar or different is still a problem that has not been sufficiently addressed. Prior studies on method naming focus on detecting linguistic anti-patterns [3], method naming bugs [4], and a multitude of naming/renaming practices [5]–[10], but do not differentiate between production and test method naming structures. For this reason, the concepts discussed in these papers may not fully generalize to test method names. This will hinder our ability to improve and support test method name quality both in the case where they are manually written by developers or automatically generated by tools. It is important to consider the unique structure of test method names to complement and increase the impact of prior work by taking into account the unique structure and purpose of test method names.

We begin addressing this problem by studying the evolution of method name structure and semantics in test suites by, primarily, analyzing the sequence of part-of-speech (POS) tags, called grammar patterns [7], associated with the method’s name. The purpose of this type of analysis is to understand how the semantics behind the test method name changes and how these semantics correlate with changes to the actual testing behavior, as defined in the code. POS tags are obtained

by splitting an identifier name into its constituent words and then annotating the split identifier manually. A grammar pattern provides us with a template-like sequence of POS tags, which are an abstract representation of an identifier’s meaning.

One problem with analyzing identifier names is that it is difficult to automatically determine the meaning of words in an identifier and how these words interact with one another. It is even more challenging to take this meaning and use it to understand how it influences, or is influenced by, the behavior of the code. Grammar patterns allow us to perform this analysis more efficiently by broadly categorizing words into their corresponding POS; this allows us to relate words together and, also, we can relate different POS tags with certain types of code behavior [7]. *The goal of this paper is to understand how test method names are structured, how they evolve in structure and meaning, and how the structure/meaning of these names relate to statically-verifiable code behavior. The data obtained in this study will be used to facilitate test name recommendation and appraisal.* We answer the following research questions:

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

RQ2: How are changes to the grammar pattern related to changes in the semantic meaning of the corresponding method name? Grammar patterns provide a way for us to learn the relationship between words, which is more granular than prior approaches, without comparing their specific definitions. This question explores how changes to the grammar pattern relate to changes in the meaning of a method name using a taxonomy, first defined by Arnaoudova et al. [10].

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

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

The contributions of this study are as follows: (1) a manually annotated dataset of test method grammar patterns (available on our website [11]), (2) new test name patterns and trends, increasing our understanding of the relationship between test name semantics and implementation, and (3) discussion of how test names evolve structurally and semantically.

II. EXPERIMENT DESIGN

Depicted in Figure 1 is an overview of our experiment design. We explain, in detail, each activity of our study in the subsequent subsections. Furthermore, the dataset utilized in this study is available on our project website [11] for extension and replication purposes.

Projects: The projects in our study consist of 800 open-source Java projects hosted on GitHub. These projects belong to a curated dataset of engineered software projects, synthesized by the Reaper tool [12]. The projects in this dataset utilize software engineering practices such as documentation, testing, and project management. Not only do we clone each repository, but we also extract commit level metadata by enumerating over the commit log of each project. The metadata we extract includes the timestamp of the commit, the author of the commit, and the files associated with each commit.

Refactorings: We utilize RefactoringMiner [13] for mining the rename refactoring operations from each project in our dataset. RefactoringMiner iterates over the entire commit history of a project in chronological order and compares the changes made to Java source code files in order to detect refactorings. RefactoringMiner is a state-of-the-art tool with a precision of 98% and a recall of 87% [14], [15]. Furthermore, we conduct our experiments on the entire commit history of the project (and not on a release-by-release comparison).

Test Suites: To identify test suites in the projects, we follow an approach similar to [16]. We first extract all Java source files (i.e., files with extension ‘.java’) that underwent a refactoring. In this study, we focus on projects utilizing the JUnit testing framework [17]. Next, using JavaParser [18], we parse the Java files by building an abstract syntax tree for each source file. We mark a file as a unit test file if the file contains JUnit import statements (i.e., `org.junit.*` or `junit.*`) and a test method. For a file to contain a unit test method, the method should have an annotation called `@Test` (JUnit 4), or the method name should start with ‘test’ (JUnit 3). In total, we detected 319,108 unit test files, out of which only 12,010 test files had undergone a *Rename Method* refactoring.

Research Question Analysis: For our research questions, we make use of the data mined/extracted by the prior activities. The activities involved in answering each research

question involve a combination of manual analysis (including data annotation) and quantitative analysis using custom-built tools/scripts. We detail our activities when addressing each research question in Section III, when reporting our results.

III. EXPERIMENT RESULTS

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

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

In this RQ, we examine the POS associated with the old and new names to identify grammar patterns that are specific to test methods. We answer this research question with three sub-RQs. The first sub-RQ looks at the frequently occurring grammar patterns that occur in test method names, while the second sub-RQ compares common grammar prefix patterns reported by prior studies, on test (by Wu and Clause [2]) and production (by Newman et al. [7]) method names, with findings from our dataset. Finally, the third sub-RQ examines how the POS changes when the method is renamed. The goal of this RQ, and therefore the sub-RQs, is to identify patterns in the way test method identifiers and their grammar patterns evolve through renames to understand how we can take advantage of this evolution in future research to provide developers with useful feedback about their renaming practices.

Approach: To understand the POS tags that constitute a method name in a unit test file, two authors manually annotated a statistically significant sample of renamed methods. In total, 632 test method rename instances (i.e., the old and new names) were manually analyzed. The analyzed sample represents the original set of 12,010 renamed methods, with a 99% confidence level and a 5% confidence interval. Similar to prior research [7], our annotation process considered 10 English POS tags— noun (N), determiner (DT), conjunction (CJ), preposition (P), noun plural (NPL), noun modifier/adjectives (NM), verb (V), verb modifier/adverbs (VM), pronoun (PR), and digit (D). Our annotation process consisted of three stages— the annotation stage, the review stage, and the discussion stage. In the first stage, each annotator annotated a set of 316 rename pairs of method names. The annotator would first split the old and new method names into their individual terms before annotating each term in the old and new names. Following the annotation process, we conducted a review stage. In this stage, the annotated datasets were exchanged between the annotators for review. If the reviewer did not agree with a specific annotation, the instance was marked for discussion. Finally, in the discussion stage, the annotator and reviewer discussed and looked at resolving conflicts. Instances where there was no consensus were discarded. During the entire process, the annotators had access to the source code and commit diff of the file containing the renamed method to refer. In total, we discarded 17 instances, leaving us with 615 annotated instances for our analysis.

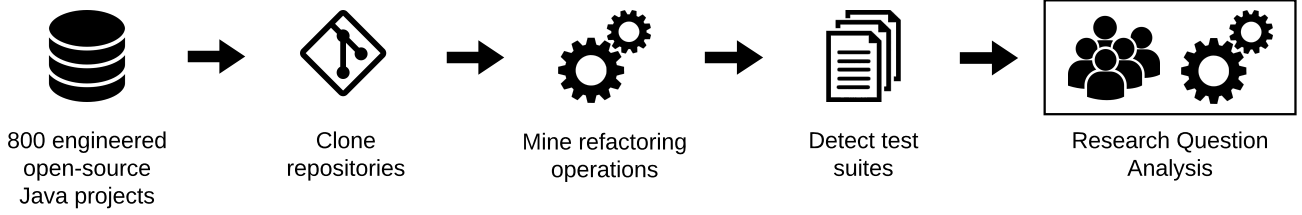


Fig. 1: Overview of our experiment design.

TABLE I: Top five frequent grammar patterns for old and new names.

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

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

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

V NM+ N is also known as a verb phrase pattern; a phrase composed of a verb followed by a noun phrase. Typically, the verb represents the action to be applied to a head-noun that exists within the same phrase; typically the rightmost noun. In test methods, we observe that the term ‘test’ frequently represents the verb. Developers utilize the noun modifier (i.e., adjective) to specify characteristics or context around the entity being tested (i.e., the entity under test). An example of this pattern is the test method `testStringEncryption` [19]. The term ‘test’ represents the verb or action of the method. The term ‘Encryption’ is the head-noun or the entity under test, while the term ‘String’ represents the noun-modifier; descriptive of the entity under test.

The next two patterns: **V N** and **V V NM N** are both derivative verb phrases, where **V N** is a verb phrase with no adjectives and **V V NM N** is a verb phrase with an extra verb. Again, the first **V** is typically the word ‘test’ or a related term (e.g., can, should; we discuss this later). An example of the **V N** pattern is `testParser` [20], the action is ‘test’, while the term ‘Parser’ represents the object the action is applied to.

The last pattern is **V**: This pattern occurs more frequently in test methods than production methods. In production code, these methods have generic names (e.g., ‘sort’) [7] since they tend to represent generic functionality. However, in test code,

the methods falling in this category are part of a test fixture¹ (i.e., a setup or teardown method). For example, the `setup` method is utilized by developers to initialize the environment for the test methods in the test suite [21].

As part of our analysis, we also look for patterns between the terms in the method’s name and statements in the method’s body. These observations we encounter can be beneficial to static analyzer based code quality tools. These include using the ‘Assert.fail’ method when the method name contains the term ‘fail’ or ‘failure’ (e.g., `failPrefixMissing` in [22]). Further, the use of the terms ‘true’ and ‘false’ in the method’s name is very likely to be associated with using the methods ‘assertTrue’ and ‘assertFalse’ in the method’s body, respectively (e.g., `testUntilTrueDefinitionOnReducedPath` in [23]). Based on results by Newman et al. [7] and our study, verb phrases (e.g., **V NM+ N**) are the most common grammar pattern for method names regardless of whether they are test or production names. Thus, in this sub-RQ, we find no significant difference in the most frequent test and production method grammar patterns. However, we also found that approximately 39.29% of test method grammar patterns are unique (i.e., they only occur once); in contrast to 24.72% of unique production method grammar patterns [7]. This difference implies that there may be a more diverse population of patterns in test methods. We address this in the next sub-RQ.

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

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

To this end, we compare the catalog of test method name patterns formulated by Wu and Clause [2] against our annotated dataset, and also examine the occurrence of these patterns in production method names discussed by Newman et al. [7].

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

TABLE II: Occurrence of test naming patterns in test and production code.

Wu and Clause's Test Pattern Name	Grammar Pattern	# of Old & New Test Method Instances	% of Test Method Instances Preserved After Rename	# of Production Method Instances	Example
Is and Past Principle Phrase	V V+	353	62%	6	<code>testGetActions</code> [24] 'test' and 'Get' are verbs
Dual Verb Phrase	V V N+	52	46%	0	<code>testFindResourceByName</code> [25] 'test' and 'Find' are verbs, while 'Resource' is a noun
Verb Phrases With(out) Prepend Test	V N V+	29	67%	3	<code>testFormUploadLargerFile</code> [26] 'test' is a verb, while 'Form' is a noun and 'Upload' is a verb
Divided Dual Verb Phrase	V N V N+	2	0	0	<code>testUidFetchBodyPeek</code> [27] 'test' and 'Fetch' are verbs, while 'Uid' and 'Body' are nouns
Noun Phrase	N	5	0	3	<code>main</code> [28] 'main' is a noun
Verb With Multiple Nouns Phrase	V N N N	–	–	0	Not observed in our dataset of annotated test methods

While we compare to Wu and Clause's work, their goals were somewhat different. Their patterns are primarily *prescriptive*; creating templates that developers should use to improve test names. Our work is *descriptive*; attempting to examine the structures latent in test names while not prescribing what developers should use. Even so, the patterns Wu and Clause create are based on testing patterns they observed, and so it is appropriate to relate our patterns to theirs. The difference between our patterns and theirs can be seen in Table II. The leftmost column contains Wu and Clause's pattern names. To its right is another column showing the grammar pattern that corresponds with Wu and Clause's named patterns. Wu and Clause abstract away some detail (i.e., the tail of our grammar patterns) to necessarily and effectively discuss general naming patterns and their semantics. In this paper, we keep these details, which causes several of our patterns to fit into a single one of Wu and Clause's patterns due to being derivative of a high-level pattern they already identified. However, this helps us understand how some of the granular differences that do not appear in Wu and Clause's work affect test name semantics.

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

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

V V N P+: This pattern is similar to Wu and Clause's 'Dual Verb Phrase' pattern. The primary difference is the presence of a preposition. For example, in the name `testReadFileFromClasspath`, 'test' and 'Read' are verbs, 'File' is a noun, and 'From' is a preposition [29]. Approximately 43% of the renames contained the prefix in the old and new names. Some of the common prepositions utilized by developers include 'of', 'with', and 'to'. Prepositions show a relationship between words, such as when and where things are related to each other. The preposition in this pattern is important because it identifies the relationship between the noun phrases on either side of it. We can use the preposition to assess the quality of a name based on which preposition is used, and whether the behavior of the test supports the use of the provided preposition. For the example above, using static analysis, we can check for the use of a file read operation that leverages the classpath. Normally, this might be difficult, but there is a finite number of prepositions in English (i.e., developers do not create new prepositions on the fly), meaning the behavior they describe is generally well-defined and finite.

N V+: This pattern consists of a noun followed by a verb (e.g., in `projectClosed`, 'project' is a noun, and 'Closed' is a verb [30]). Looking at the set of production methods, we observe ten instances of methods with this prefix pattern. From our annotated dataset of test methods, we observe 22 rename instances of this prefix. Additionally, approximately 64% of the renames contained the prefix in the old and new names.

+VM+: While not strictly a prefix grammar pattern, we include this observation in our findings since 1) verb modifiers have not been discussed at length in prior literature, 2) we found several patterns containing adverbs in our dataset, and 3) it is possible to use some of our observations about naming and implementation practices based on the presence or absence of certain adverbs. We start with an example: in the name `test_get_NotExisting`, 'Not' is an adverb [31]). We encounter 86 rename instances containing one or more adverbs in the name. Furthermore, we notice that developers utilize the same adverb from the old name in

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

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

the new name when performing a rename of the method 78% of the time. Additionally, the top three terms associated with an adverb are ‘not’ (26 instances), ‘when’ (25 instances), and ‘exactly’ (5 instances). When combined with static code analysis, our observation becomes useful as it helps in appraising the name of an identifier. For instance, when examining the source code, we observe that method names containing the adverb ‘not’ are typically associated with some form of null based checking (e.g. use of ‘assertNull’ in the method `test_get_NotExisting` [31] and the use of ‘assertNotNull’ in the method `deleteindexNotExists` [32]). Finally, looking at production methods, we encountered seven instances of methods using this POS within its name.

+DT+ : Our rationale for the analysis of determiners is similar to our analysis of the **+VM+** pattern. Our dataset contains 72 instances that contain determiners in either the old or new name. From this set, there are 42 instances where the developer uses the same determiner in the old and new name (e.g., the term ‘All’ is preserved in the rename `findAllWithGivenIds` $\rightarrow$ `findAllWithIds` [33]). Regarding terms, the top three popular determiners are ‘the’, ‘no’, and ‘all’. In terms of static code analysis, we observe that the term ‘all’ frequently co-occurs with collection-based data types in the method body (e.g., the use of ‘List<Long>’ in the method `testExecuteAll` [34]). The static analysis accuracy can be further improved by incorporating Peruma et al. [35] findings, specifically the findings on collection-based data types and singular/plural term changes. Using grammar patterns, we have confirmed the existence of several naming patterns introduced by Wu and Clause. In addition, we identify patterns that were not identified in Wu and Clause’s original set of patterns. The patterns we present are not frequent in production method names based on prior research, indicating that they are specific to test method names.

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

In this sub-RQ, we examine the evolution of grammar patterns (i.e., the change in the grammar pattern when a method is renamed). In summary, our annotated dataset of 615 rename instances contained 168 (or approximately 27.32%) rename instances that did not show a change in grammar (i.e., the old and new grammar patterns were the same). Represented in Table III are the top five frequently occurring complete grammar pattern pairs. However, looking at the number of instances associated with each pair, we observe a low count (the most being 14 instances). Furthermore, our dataset contained 446

instances of grammar pattern pairs that occurred only once. This phenomenon (i.e., a wide variety of grammar patterns) highlights the diversity of our dataset and, therefore, impacts our analysis of rename pairs. Therefore, for the same purpose as RQ 1.2 we use prefix patterns to perform our analysis. We extracted frequently occurring pairs of rename prefixes for patterns where either the old or new name consists of prefixes of length two, three, four, or five. From this data, we show the top three frequently occurring pairs in Table IV; the complete listing is available on our website [11].

From these tables, we make a couple of observations. The first is that renames do not typically change the POS tag of a word. Even when a word is changed, it is still the same type (i.e., at the POS level). Further, these renames follow the typical verb phrase method naming grammar pattern $V NM N \rightarrow V NM N$. For example, in `commit` [19], when renaming the method `testStringEncryption` $\rightarrow$ `testStrongEncryption`, the POS is preserved even though terms in the name are changed; the terms ‘String’ and ‘Strong’ are considered as noun modifiers in this instance.

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

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

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

Approach: To determine the semantic change a name undergoes during a rename, we utilize a rename taxonomy defined by Arnaudova et al. [10] and utilized in prior identifier rename studies [5], [6], [35] on our annotated dataset. This taxonomy helps us categorize renames into two categories—renaming form and semantic change. The renaming form looks at the terms added and removed to determine the complexity of the rename—simple, complex, reordering, and formatting. A rename is simple if only one term is added or removed.

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

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

TABLE V: Top five frequently occurring rename semantic updates for pairs of complete grammar patterns.

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

A complex change occurs if more than one term is added or removed. Reordering is when two or more terms switch positions. Finally, a formatting change occurs if the developer only makes a change in case or adds/removes a separator or number. In terms of semantic change categories, a rename can either preserve or modify the meaning of the name. A modification to a name can change, narrow, broaden, add or remove the meaning of the name.

From our set of 615 annotated instances, we observe 291 (or approximately 47%) of the instances had a simple change, while 261 instances (or approximately 42%) had a complex change. From the semantic category, as depicted in Figure 2, we observe 255 (or approximately 41%) of instances had a change in meaning, while the narrowing and broadening category each had approximately 18%. These findings are in contrast to prior research [5], [6], which shows that the majority of renames are of simple form and narrow in meaning. The prior studies utilize datasets comprising of mined rename refactoring operations of test and production source code files in Java projects. Looking at the set of renames categorized under preserve, we observe that the majority of these renames are due to developers either adding or removing numbers or underscore characters to/from the old name or performing a change of case (e.g., `test_13` → `test13` [36]).

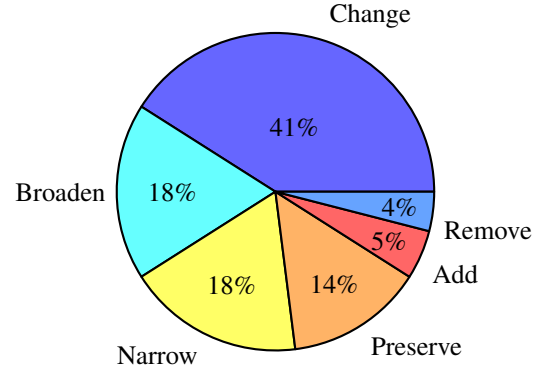


Fig. 2: Proportion of semantic updates to 615 annotated test methods.

TABLE VI: Frequently occurring rename pairs of prefix patterns for different semantic categories.

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

Examining the change in meaning instances, 208 (or 33.82%) of the instances show an unrelated relationship between the old and new names. For example, in renaming `testLog` $\rightarrow$ `testEigenSingularValues` [37], there is no semantic relationship between the swapped terms. Approximately 7.15% instances contained more than one type of relationship between the old and new names. For example, in renaming `testDeserializeExpandCharge` $\rightarrow$ `testDeserializeWithExpansions` [38], we observe an addition and removal of terms as well as a change in plurality. Finally, three (or 0.49%) instances exhibited an antonym relationship as in case of renaming the method `shouldAcceptRaxProtocols` $\rightarrow$ `shouldRejectRaxProtocols` [39]; here we see the term ‘Reject’ replacing ‘Accept’.

In Table V, we provide the top five rename forms and semantic updates associated with a complete grammar pattern pair. From this table, we observe that the most frequent grammar pair, $V\ NM\ N \rightarrow V\ NM\ N$ is mostly associated with a change in meaning. For example, in the rename `commit` [19], `testStringEncryption` $\rightarrow$ `testStrongEncryption`, a single term is replaced making it a Simple form type change and since there is no semantic relationship between the terms ‘String’ and ‘Strong’ it is categorized as a general change in meaning. However, from this table, we observe a low volume of instances of grammar patterns associated with the semantic categories; this behavior is similar to what is observed in RQ 1.3. Hence, similar to RQ 1.3, going forward, we look at the relationship between prefix grammar patterns and name semantics. Presented in Table VI, we provide the top three frequently occurring prefix patterns for each semantic category. The complete listing is available on our website [11].

From Table VI, we observe that the rename prefix pattern $V\ V \rightarrow V\ V$ is associated with all semantic categories. However, it is more prevalent with the change in meaning category. This same prefix pattern is also the most common rename pattern, as reported in RQ 1.3. From the table, we observe that remove and broaden meaning shows a divergence in the prefix pattern; the most frequently occurring prefix pattern for these two categories is $V\ V \rightarrow V\ NM$. For the broadening pattern, we observed that in the majority of developers tend to remove the term ‘test’ from the old name (e.g., `testPinnedExternals` $\rightarrow$ `pinnedExternals` [40]). As the number of prefixes increases, the volume of these instances being associated with a semantic category decreases. Again, similar to RQ 1.3, this phenomenon will help determine the quality of a test method’s name either when a developer performs a rename or during static analysis of code. However, as these are prefix patterns, it should be noted that terms associated with the POS tags in the prefix might not always be the terms contributing to the semantic transformation of the method name. Hence, the findings from this RQ should be used in conjunction with findings from other RQs and also prior work, such as Peruma et al. study of identifier renaming using data types and co-occurring refactorings [35].

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

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

Approach: In this RQ, we examine the frequent terms added to and removed from test methods due to a rename. The experiment in this RQ utilizes the complete dataset of test method names. We first utilize the heuristic splitter algorithm implemented in the Spiral package [41] to determine the terms that form a name. Next, for each rename instance, we extract only the terms that were added and removed. Finally, for each added and removed pair of terms, we count the number of times the pair exists in the dataset. For example, when `getEmployeeName` is renamed to `testEmployeeLastName`, the added terms are ‘test’ and ‘Last’, while the removed term is ‘get’. We search our dataset for the occurrence of ‘test’ & ‘get’ and ‘Last’ & ‘get’. Additionally, as part of our qualitative approach, two of the authors manually analyzed a statistically significant sample that comprises of the top 646 frequently occurring pairs. The sample represents a 99% confidence level and a 5% confidence interval from our population of 21,615 pairs of added and removed terms. As part of this analysis, the authors annotated the semantic relationships between the added and removed terms. The semantic annotations include: synonyms, antonyms, specializations, and generalizations.

Analyzing the list of 646 removed-added pairs, we observe instances where the developer either adds or removes numerical digits to or from the replacement term. An in-depth look at these identifiers shows that a vast majority of such names usually do not contain any other terms that describe the behavior of the test method (e.g., `test15_6_5` $\rightarrow$ `test16_9_5` [42]). Most likely, these are auto-generated tests or tests utilized for debugging purposes. To facilitate the use of English semantic rules to determine the relationship between the term pairs, our analysis of term pairs will be limited to only pairs that do not have numerical digits. Looking

at the top five frequently occurring term pairs, we observe developers replace ‘has’ with ‘contains’ (94 instances), ‘test’ with ‘can’ (58 instances), ‘all of’ with ‘at least’ (46 instances), ‘with’ with ‘when’ (40 instances), and ‘test’ with ‘should’ (38 instances). Additionally, we also observe that developers frequently replace the term ‘test’ with a term associated with a Boolean return type (e.g., ‘can’, ‘is’, ‘should’).

Next, we look at the different types of semantic relationships between the removed-added pairs. From our dataset, we observe that 294 of the removed terms were replaced with terms of the same POS. For example, in renaming `testFilterBaseNice` → `testSelectBaseNice` [43] the developer replaces the term ‘Select’ with ‘Filter’, both of which are verbs. The majority of replacement terms were added in the same position as the removed term in the name. Looking at the types of semantic relationships in the dataset, we observe 36 pairs of terms as synonyms (e.g., `boundingCube` → `boundingBox` [44]) and 22 pairs having an antonym relationship (e.g., `genericExtension` → `specificExtension` [45]). We also identified 12 instances each of specialization (e.g., `testPredictions` → `validatePredictions` [46]) and generalization (e.g., `listContains` → `collectionContains` [47]).

Looking at the root (i.e., stem) of the removed-added term pairs, we observe that 70 pair instances have the same stem. For example, in the following rename `testTwippleUploader` → `testTwippleUpload` [46], the terms ‘Uploader’ and ‘Upload’ have the same stem-‘upload’. We also observe 17 instances of tense change (e.g., `isOrderedFailure` → `isInOrderFailure` [48]) and nine instances of plurality changes (e.g., `enqueueJob` → `enqueueJobs` [49]), and spelling corrections (e.g., `projectVisitorIsInkvoled` → `projectVisitorIsInvoked` [50]) each.

Our manual analysis shows that determining the relationship between terms is a challenging task due to the diverse ways in which developers rename identifiers. We made the following observations during our qualitative analysis: (1) although proper naming helps understand what the test verifies and how the underlying system behaves, some terms are ambiguous, which makes it challenging to determine the semantic relationship between the pair due to the use of domain terminology (e.g., ‘LBDevice’ is replaced with ‘Zeus’ [51]), (2) multiple terms can replace a single term and vice versa; this type of change is done due to specialization/generalization of behavior or in situations where the names are synonyms (e.g., the terms ‘not started’ are replaced by ‘closed’ [52]), and (3) the terms are unrelated (e.g., ‘Latency’ is replaced with ‘Metrics’ [53]). Finally, when examining the code, we observe that specific terms in a method’s name can indicate the presence of specific statements in the body of the method. For instance, we observe that the presence of the terms ‘at least’, ‘all of’ or ‘all’ acts as a sign that a method performs tests on collection based objects such as List, Map, or custom collection types. For example, the method `findAllWithGivenIds` contains a collection object that is subject to a series of tests (i.e.,

assertion statements). Similarly, the occurrence of the term ‘exception’ indicates that the purpose of such methods is to verify that an exception occurs as part of the execution of the test. In such instances, developers either utilize the ‘expected’ parameter as part of the Test annotation or places an assertion statement in the catch section of the try-catch block that handles the exception that the developer expects to be thrown (e.g., `invokingStaticMethodQuietlyShouldWrap IllegalArgumentException` [54]). These observations show how static analysis combined with NLP techniques can support the automation of identifier name appraisal algorithms.

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

IV. RELATED WORK

We divided our reporting of related work into three areas—studies that explored the naming of test methods, studies that investigated grammar patterns in identifier names, and studies around the renaming of identifiers in source code.

A. Test Method Names

Using natural language techniques, Zhang et al. [1], parse test method names in order to generate templates for the test methods automatically. In their approach, the authors utilize the information contained within the name of a test method—the action phrase and the predicate phrase. To perform the parsing, the authors depend on English grammar constructs. The authors achieve an accuracy of over 80% for their template generation approach. In a subsequent study [55], the authors present an approach and tool, *NameAssist*, to generate descriptive names for test methods based on the body of the test method. In their approach, the authors analyze the statements within the test method to determine the action, expected outcome, and scenario under test. Based on this static analysis, the authors utilize natural language processing techniques to generate a descriptive name for the test method. Daka et al. [56], propose an approach to generate short descriptive test method names based on API-level coverage goals; the authors validate their approach by surveying 47

students. Lin et al. [57] investigate the quality of identifiers in test suites and perform a comparison against production identifiers. Results from the survey show that identifiers in test suites are of poor quality, with automatically generated test suites demonstrating even more quality concerns. Further, a comparison of rename recommendation tools shows that they perform poorly on test suites. Wu and Clause [2] provide an approach to identify non-descriptive test method names and provides developers with information for a more descriptive name. Their approach depends on a set of test patterns. From these patterns, their mechanism extracts the action, predicate, and scenario from the current name of the test and body of the test method. By comparing the extracted information, their approach determines if the current test name is descriptive.

B. Identifier Grammar Patterns

In their study of grammar patterns in identifier names, Newman et al. [7] observe a set of grammar patterns developers utilize to describe program behavior. Some of their observations include: noun phrases are one of the most common grammar patterns, function identifiers are more likely to be represented by a verb phrase, and collection type frequently utilize a plural head-noun. Additionally, the authors indicate that the current POS taggers are not effective on source code identifiers. In an empirical study on 5,000 open-source projects, Zhang et al. [58] observe that nouns, verbs, and adjectives are three of the most common POS tags utilized by developers in crafting identifier names. The authors utilize Stanford Parser to parse the POS tags from an identifier’s name automatically. Binkley et al. [59] investigate the effectiveness of Stanford Log-linear POS Tagger on field names. Through this study, the authors propose four rules, based on POS tags, for improving field names. A study of naming in multiple programming languages by Liblit et al. [60] shows how natural language influences the use of words in these languages. Høst and Østfold [4] examine unusual method names and propose a set of naming rules to uncover issues in method names. The authors utilize POS tags along with the return type, control flow, and parameters of the method to detect naming violations based on a set of rules.

C. Identifier Renaming

In a study on identifier renames, Arnaoudova et al. [10] propose a semantic taxonomy for classifying identifier renames. Additionally, through a developer survey, the authors observe that developers confirm that identifier renaming is a challenge. Studies around contextualizing identifier renaming by Peruma et al. [5], [6], [35] show that the majority of identifier renames are performed with the intent of narrowing the meaning of the identifier name. The authors also observe that there is a subset of refactorings that occur before a rename refactoring. Additionally, when looking at data type changes, the authors observe instances where developers change the plurality of a name in response to its type changing to/from a collection. Finally, the authors also discuss challenges around analyzing rename refactorings and commit messages.

Work on identifying rename opportunities by Allamanis et al. [61], [62] uses statistical language models to mine natural source code naming conventions. The authors approach looks for potential variable, method, and class renaming opportunities. Research by Liu et al. [63] look at recommending renames based on the prior rename activities performed by developers on the source code. Additionally, by studying the relationship between argument and parameter names, the authors develop an approach to detect naming anomalies and suggest renames to developers [64]. In their study, Jiang et al. [65] observe that the effectiveness of *code2vec*, a machine learning-based approach for method name recommendations, fails in a realistic setting. The authors also propose a heuristic-based approach that outperforms *code2vec*.

V. THREATS TO VALIDITY

The projects in our dataset are limited to open-source Java systems, and the results may not generalize to systems written in other languages. However, these projects are from a set of engineered Java systems [12] and have been utilized in prior refactoring related studies [35]. Furthermore, since the test files in our dataset belong to a variety of projects, our analysis is based on test methods implemented by different developers and thereby representative. While there are other tools available to mine refactoring operations, Refactoring-Miner outperforms the other tools [66] and is frequently utilized in refactoring studies [67]–[70]. Our analysis of test files is limited to projects utilizing the JUnit testing framework, and hence the results might not generalize to other testing frameworks. However, prior unit testing based research has frequently focused on JUnit, such as in the case of test smells [71]. The findings of our research questions are based on a sample set of annotated data. To ensure unbiased representativeness, our sample size is a statistically significant random sample, and our annotation process included a review phase.

VI. DISCUSSION & CONCLUSION

In this paper, we examine how developers craft method names in test suites. We made it our goal to understand how test method names are structured, how they evolve in structure and meaning, and how the structure/meaning of these names relate to statically-verifiable code behavior. Our findings show the effectiveness of using grammar patterns to understand naming practices, and how those practices translate to statically-verifiable code behavior. In this section, we discuss how the findings from our RQ’s and observations align with the study’s goals, along with the challenges and future research directions.

A. Takeaways

Takeaway 1: *Test names have a structure that differs from production names. Some of this structure can be leveraged to provide test-specific recommendations*

From RQ1, we observe that test method names vary in the number of terms making up the name and the POS tags associated with these terms. During our manual annotation, we observe that test method names are highly specific to their

intended behavior. This occurrence is not surprising as the purpose of test methods is to test/exercise the atomic units of the production code [72]; test suites can have more than one test method to test the behavior of a single production method. Hence, developers craft test names to be as descriptive as possible, going as far as even to describe conditions that appear within the method using adverbs (e.g., not) and prepositions (e.g., after) in the test name. The use of these specific adverbs and prepositions is interesting because the appearance of these words correlates with the appearance of specific structures within the code; the words and code structure are semantically related in ways that can be statically detected. This correlation allows for recommendations to be made to developers based on either the presence of certain words in the identifier name or the presence of code structures in the function. The descriptive naming we observe aligns with the action phrase and the predicate phrase naming pattern [1]. This phenomenon is highlighted by our findings that show how common prefix grammar patterns for test method names are not common patterns for production methods.

Takeaway 2: *Some of the prefixes detected in our dataset indicate the existence of additional test name patterns*

While our findings from RQ1 confirm the grammar patterns identified by Wu and Clause, we also observed test naming patterns for test method names. These grammar patterns are: $V V N P+$ and $N V+$. Of these two, $V V N P+$ is the only one whose behavior is not sufficiently described by patterns present in prior work, which does not mention prepositions. As stated earlier, prepositions correlate with specific types of test behavior. Thus, we feel that this pattern is both legitimate and new amongst the naming patterns presented in prior research; differentiating this pattern from Wu and Clause’s $V V N$ is important because it implies different behavior, which a recommendation system should be aware of. In addition, like with prepositions, we find that the existence of adverbs (VM) and determiners (DT) in test method names correlate with specific code behaviors. This has not been explored in prior work and is novel to our investigation. The evidence we empirically find suggests that the relationship between these POS tags and code behavior can be used to detect and recommend the removal of linguistic anti-patterns.

Takeaway 3: *Test method name refactorings tend to change the meaning of terms in the name*

By mining rename refactoring operations in projects, we extract renames performed by developers on test methods. This data provides us with the opportunity to study the evolution of method names. From RQ2, we observe that unlike prior research on all types of method names, not focusing exclusively on test suites [5], [6], our study shows that developers frequently change the meaning of test names more often than narrowing the meaning. This indicates one of two situations: either test method names evolve by significantly altering the meaning of words within the name, or it is more challenging to analyze the relationship between words in test methods using approaches such as WordNet than it is to analyze word

relationships on other types of methods as was done in prior work [5], [6], [8]. Whichever case we are in, it is clear that the relationship between words in test methods as they evolve is different from the general population of methods. Thus, recommending test name structure or words will potentially require specialized approaches trained specifically on test naming structures. More research is needed in this area to understand this phenomenon.

Takeaway 4: *There are common words and phrases which are synonymous or metonymous in test method renames*

RQ3 highlights the five most frequently swapped terms in method names during rename operation— with ‘has’ and ‘contains’ being the most common term pair. These terms/phrases are synonymous, or metonymous, with one another. Further, like the adverb and preposition examples from RQ1, they can be directly linked to specific behaviors that appear within the test code. Thus, it is possible to use these patterns to appraise test names and their contents to provide recommendations. We were only able to identify a handful of these patterns, but they represent a strong start to potential future recommendations and a path toward finding other similar patterns. One clear future step for these terms is to begin taking advantage of the synonym/metonym relationship between them to increase naming consistency in test methods or recommend changes to the test body based on the wording in the test name. This direction is similar to the idea of method name debugging [4] and linguistic anti-patterns [3].

Summary. Given the goal of our paper, we have confirmed the usefulness of grammar patterns in understanding test method name structure, how this structure evolves, and how it relates to code behavior. The results from our work help confirm the need for test-specific naming support and provide several recommendations in terms of what kind of support can be readily provided. Specifically, we show that certain types of words are frequently used in the context of statically-verifiable code behaviors. While further research into this phenomenon is required, our work is a strong starting point; providing both the evidence for the existence of these patterns and the means through which these patterns can be further explored, detected, and leveraged for recommendation.

B. Challenges

Key challenges in utilizing standard NLP techniques in studies like ours include parsing of misspellings (e.g., ‘get’ is misspelt as ‘het’ in `hetInput` [73]), contractions (e.g., expand ‘dont’ to ‘do not’ [74]), domain/technology terms (i.e., terms not part of general purpose ontologies), and preamble terms or terms at the start of the name to indicate a specific action/purpose (e.g., use of ‘Ignore’ in `IGNOREtestHttpsCheckOut` to exclude the test [75]). Additionally, analyzing the code and comments surrounding a method will help decide the correct POS tag for terms in its name.

VII. ACKNOWLEDGEMENTS

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

REFERENCES

- [1] B. Zhang, E. Hill, and J. Clause, “Automatically generating test templates from test names (n),” in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 506–511, 2015.
- [2] J. Wu and J. Clause, “A pattern-based approach to detect and improve non-descriptive test names,” *Journal of Systems and Software*, vol. 168, p. 110639, 2020.
- [3] V. Arnaoudova, M. Di Penta, and G. Antoniol, “Linguistic antipatterns: what they are and how developers perceive them,” *Empirical Software Engineering*, vol. 21, 01 2015.
- [4] 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.
- [5] A. Peruma, M. W. Mkaouer, M. J. Decker, and C. D. Newman, “An empirical investigation of how and why developers rename identifiers,” in *Proceedings of the 2Nd International Workshop on Refactoring, IWorR 2018*, (New York, NY, USA), pp. 26–33, ACM, Sept. 2018.
- [6] A. Peruma, M. W. Mkaouer, M. J. Decker, and C. D. Newman, “Contextualizing rename decisions using refactorings and commit messages,” in *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 74–85, Sep. 2019.
- [7] C. D. Newman, R. S. AlSuhaibani, M. J. Decker, A. Peruma, D. Kaushik, M. W. Mkaouer, and E. Hill, “On the generation, structure, and semantics of grammar patterns in source code identifiers,” *Journal of Systems and Software*, vol. 170, p. 110740, 2020.
- [8] J. Pantiuchina, F. Zampetti, S. Scalabrino, V. Piantadosi, R. Oliveto, G. Bavota, and M. D. Penta, “Why developers refactor source code: A mining-based study,” *ACM Trans. Softw. Eng. Methodol.*, vol. 29, Sept. 2020.
- [9] K. Liu, D. Kim, T. F. Bisseyandé, T. Kim, K. Kim, A. Koyuncu, S. Kim, and Y. Le Traon, “Learning to spot and refactor inconsistent method names,” in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 1–12, 2019.
- [10] V. Arnaoudova, L. M. Eshkevari, M. D. Penta, R. Oliveto, G. Antoniol, and Y. Guéhéneuc, “Repent: Analyzing the nature of identifier renamings,” *IEEE Transactions on Software Engineering*, vol. 40, no. 5, pp. 502–532, 2014.
- [11] <https://scanl.org/>.
- [12] 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.
- [13] 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), p. 483–494, Association for Computing Machinery, 2018.
- [14] 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*, Association for Computing Machinery, 2016.
- [15] C. Vassallo, G. Grano, F. Palomba, H. C. Gall, and A. Bacchelli, “A large-scale empirical exploration on refactoring activities in open source software projects,” *Science of Computer Programming*, vol. 180, 2019.
- [16] A. Peruma, K. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba, “On the distribution of test smells in open source android applications: An exploratory study,” in *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering, CASCON ’19*, (USA), p. 193–202, IBM Corp., 2019.
- [17] “JUnit.” <https://junit.org/>.
- [18] “Javaparser.” <http://javaparser.org/>.
- [19] “apache/commons/compress/archivers/zip/generalpurposebittest.java.” <https://github.com/apache/commons-compress/commit/fa2e5bd>.
- [20] “src/test/java/org/scrabble/projection/protocolprojectiontest.java.” <https://github.com/scrabble/scrabble-java/commit/62d9cdb>.
- [21] “src/test/java/com/github/koraktor/mavanagaiata/abstractgitmojotest.java.” <https://github.com/koraktor/mavanagaiata/commit/27f52ab>.
- [22] “domain/dependence/checkmethodtest.java.” <https://github.com/socialsoftware/blended-workflow/commit/92a9539>.
- [23] “unittests/countexampleuntilunittest.java.” <https://github.com/hhu-stups/prob-rodinplugin/commit/c3f554b>.
- [24] “/src/com/eclipsesource/tabris/internal/ui/remotepagetest.java.” <https://github.com/eclipsesource/tabris/commit/df56d08>.
- [25] “src/test/java/org/rethub/web/controller/testresourcecontroller.java.” <https://github.com/rethub/rethub-spring-stack/commit/c55d122>.
- [26] “src/test/java/io/vertx/core/http/httpptest.java.” <https://github.com/eclipse-vertx/vert.x/commit/cdc8172>.
- [27] “test/org/freenetproject/freemail/imap/imapuidfetchtest.java.” <https://github.com/freenet/plugin-freemail/commit/778a842>.
- [28] “src/test/java/org/apache/hama/bsp/message/testavromessagemanager.java.” <https://github.com/apache/hama/commit/a5483d1>.
- [29] “src/test/java/io/vertx/test/core/fileresolvertest.java.” <https://github.com/eclipse-vertx/vert.x/commit/f5bfd8c>.
- [30] “core/internal/proportions/projectchangeintegrationspdetest.java.” <https://github.com/usus/usus-plugins/commit/0a66ccd>.
- [31] “src/test/java/javax/cache/cachetest.java.” <https://github.com/jsr107/jsr107tck/commit/27149d0>.
- [32] “io/searchbox/indices/deleteindexintegrationtest.java.” <https://github.com/searchbox-io/jest/commit/0c5346a>.
- [33] “Simplegemfirerepositoryintegrationtests.java.” <https://github.com/spring-projects/spring-data-gemfire/commit/15aae69>.
- [34] “Getmetrictreamscopeidsdbmapptest.java.” <https://github.com/lmco/eurekastreams/commit/9d456b6>.
- [35] A. Peruma, M. W. Mkaouer, M. J. Decker, and C. D. Newman, “Contextualizing rename decisions using refactorings, commit messages, and data types,” *Journal of Systems and Software*, vol. 169, p. 110704, 2020.
- [36] “src/test/java/net/sourceforge/pmd/lang/java/dfa/acceptancetest.java.” <https://github.com/adangel/pmd/commit/e280151>.
- [37] “src/test/org/apache/hama/matrix/testsingularvaluedecomposition.java.” <https://github.com/apache/hama/commit/6ba4dc3>.
- [38] “src/test/java/com/stripe/model/issuerfraudrecordtest.java.” <https://github.com/stripe/stripe-java/commit/42e8cec>.
- [39] “atlas/rax/api/validation/validator/raxloadbalancervalidatortest.java.” <https://github.com/openstack-atlas/atlas-lb/commit/3bc61ec>.
- [40] “src/test/java/hudson/scm/subversionscmtest.java.” <https://github.com/jenkinsci/subversion-plugin/commit/179fec8>.
- [41] M. Hucka, “Spiral: splitters for identifiers in source code files,” *Journal of Open Source Software*, vol. 3, no. 24, p. 653, 2018.
- [42] “src/test/java/org/neo4j/cypherdsl/cypherreferencetest.java.” <https://github.com/neo4j-contrib/cypher-dsl/commit/4ae0202>.
- [43] “src/test/java/org/linqs/psl/model/rule/groundruletest.java.” <https://github.com/linqs/psl/commit/bab277d>.
- [44] “test/georegression/geometry/testutilpoint3d_f64.java.” <https://github.com/lessthanoptimal/georegression/commit/ed24838>.
- [45] “test-src/net/dougqh/jak/jvm/assembly/api/genericstest.java.” <https://github.com/dougqh/jak/commit/7d4cba0>.
- [46] “test/de/jungblut/classification/nn/multilayerperceptrontest.java.” <https://github.com/thomasjungblut/thomasjungblut-common/commit/2bce452>.
- [47] “src/test/java/org/junit/contrib/truth/collectiontest.java.” <https://github.com/google/truth/commit/2b57a43>.
- [48] “src/test/java/com/google/common/truth/iterablesubjecttest.java.” <https://github.com/google/truth/commit/5de3d21>.
- [49] “src/test/java/net/greghaines/jesque/integrationtest.java.” <https://github.com/gresrun/jesque/commit/3f6a680>.
- [50] “java/org/pitest/pitclipse/pitranner/config/pitexecutionmodetest.java.” <https://github.com/pitest/pitclipse/commit/751b3d7>.
- [51] “atlas/api/validation/validators/networkitemvalidatortest.java.” <https://github.com/openstack-atlas/atlas-lb/commit/d7d7f87>.
- [52] “src/test/java/org/jsr107/tck/cachetest.java.” <https://github.com/jsr107/jsr107tck/commit/c2d9369>.
- [53] “src/test/java/stormpot/configtest.java.” <https://github.com/chrisvest/stormpot/commit/b4eeba8>.
- [54] “src/test/java/joptsimple/internal/reflectiontest.java.” <https://github.com/jopt-simple/jopt-simple/commit/cf6d097>.
- [55] B. Zhang, E. Hill, and J. Clause, “Towards automatically generating descriptive names for unit tests,” in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016*, (New York, NY, USA), p. 625–636, Association for Computing Machinery, 2016.
- [56] E. Daka, J. M. Rojas, and G. Fraser, “Generating unit tests with descriptive names or: Would you name your children thing1 and thing2?,” in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2017*, (New York, NY, USA), p. 57–67, Association for Computing Machinery, 2017.

- [57] B. Lin, C. Nagy, G. Bavota, A. Marcus, and M. Lanza, "On the quality of identifiers in test code," in *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 204–215, 2019.
- [58] J. Zhang, S. Liu, J. Luo, J. Liang, and Z. Huang, "Exploring the characteristics of identifiers: A large-scale empirical study on 5,000 open source projects," *IEEE Access*, vol. 8, pp. 140607–140620, 2020.
- [59] 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), p. 203–206, Association for Computing Machinery, 2011.
- [60] 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.
- [61] 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, FSE 2014*, (New York, NY, USA), p. 281–293, Association for Computing Machinery, 2014.
- [62] 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), p. 38–49, Association for Computing Machinery, 2015.
- [63] 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.
- [64] 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 *Proceedings of the 38th International Conference on Software Engineering, ICSE '16*, (New York, NY, USA), p. 1063–1073, Association for Computing Machinery, 2016.
- [65] L. Jiang, H. Liu, and H. Jiang, "Machine learning based recommendation of method names: How far are we," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 602–614, 2019.
- [66] L. Tan and C. Bockisch, "A survey of refactoring detection tools," in *Software Engineering*, 2019.
- [67] A. Peruma, "A preliminary study of android refactorings," in *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pp. 148–149, 2019.
- [68] E. A. AlOmar, A. Peruma, C. D. Newman, M. W. Mkaouer, and A. Ouni, "On the relationship between developer experience and refactoring: An exploratory study and preliminary results," in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops, ICSEW'20*, (New York, NY, USA), p. 342–349, Association for Computing Machinery, 2020.
- [69] A. Peruma, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba, "An exploratory study on the refactoring of unit test files in android applications," in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops, ICSEW'20*, (New York, NY, USA), p. 350–357, Association for Computing Machinery, 2020.
- [70] E. A. AlOmar, A. Peruma, M. W. Mkaouer, C. Newman, A. Ouni, and M. Kessentini, "How we refactor and how we document it? on the use of supervised machine learning algorithms to classify refactoring documentation," *Expert Systems with Applications*, p. 114176, 2020.
- [71] V. Garousi and B. Küçük, "Smells in software test code: A survey of knowledge in industry and academia," *Journal of Systems and Software*, vol. 138, pp. 52 – 81, 2018.
- [72] R. Pressman and D. Bruce R. Maxim, *Software Engineering: A Practitioner's Approach*. McGraw-Hill Education, 2014.
- [73] "src/test/java/org/richfaces/component/ajaxvalidationtest.java." <https://github.com/richfaces4/components/commit/235d8e9>.
- [74] "packetprocessor/iq/namespace/register/unregistersettest.java." <https://github.com/buddycloud/buddycloud-server-java/commit/39ec30f>.
- [75] "src/test/java/hudson/scm/subversionscmtest.java." <https://github.com/jenkinsci/subversion-plugin/commit/179fec8>.