On The Quality of Identifiers in Test Code

Bin Lin*, Csaba Nagy*, Andrian Marcus†, Gabriele Bavota*, Michele Lanza*
*Software Institute @ Università della Svizzera italiana (USI), Switzerland †University of Texas at Dallas, USA

***Abstract*—Meaningful, expressive identifiers in source code can enhance the readability and reduce comprehension efforts. Over the past years, researchers have devoted considerable effort to understanding and improving the naming quality of identifiers in source code. However, little attention has been given to test code, an important resource during program comprehension activities.**

To better grasp identifier quality in test code, we conducted a survey involving manually written and automatically generated test cases from ten open source software projects. The survey results indicate that test cases contain low quality identifiers, including the manually written ones, and that the quality of identifiers is lower in test code than in production code. We also investigated the use of three state-of-the-art rename refactoring recommenders for improving test code identifiers. The analysis highlights their limitations when applied to test code and supports mapping out a research agenda for future work in the area.

***Index Terms*—Test code quality; Empirical study**

I. Introduction

Identifiers represent a major part of the source code [1] and program comprehension becomes significantly harder when they are not meaningful [2], [3]. Indeed, while comprehending code, programmers rely on the meaning encoded in names [4], since those are supposed to record knowledge and communicate key concepts in the source code [1], [5]. Poor identifier names can hinder code comprehension and negatively affect code quality [6]. Moreover, they may also threaten the performance of identifier based software engineering tools [7], [8].

Consequently, many naming conventions, guidelines, and best practices have been distilled to help developers to choose appropriate names for their identifiers. For example, the Java Language Specification [9] indicates rules for naming local variables and parameters: *e.g.,* "*should be short, yet meaningful*"; "*one-character identifiers should be avoided, except for temporary and looping variables, or where a variable holds an undistinguished value of a type*". Researchers have also extensively studied what makes an identifier good or bad [2]–[4], [10], [11], and how it is possible to automatically improve existing ones using Natural Language Processing (NLP) [12], thesauruses [13], or statistical language models [14], [15].

Existing empirical studies and rename refactoring techniques target the source code as a whole when studying/improving identifier names, often ignoring the test code, despite its important peculiarities. For instance, many studies found that developers take less care of the quality of test code as compared to production code, thus leading to possible quality issues in the tests [16]–[21], including specific types of smells [22]–[24] accompanied by refactorings aimed at removing them [22].

The quality problem of test code is further exacerbated when using automated test suite generators [25], [26]. These tools [27] represent a useful aid to identify defects through a systematic, automatic approach and to improve the coverage of a test bed. Another possible use case is to generate an initial test suite and then manually improve/evolve it. In any case, the generated code, and especially the assertions of tests, need to be manually validated. Hence, the quality of the generated code matters, including the meaningfulness of the used identifiers.

We first present an empirical investigation of the quality of identifiers in test code and compare it to the quality of production code. Given the result that the identifier quality is often unsatisfactory, especially for the test code, we investigate whether the identifier quality can be improved by three state-of-the-art rename recommenders [15], [28], [29]. More specifically, in this paper we address the following research questions:

RQ_1: What is the quality of identifiers in the test code of open source projects? We conducted a survey asking 19 participants to inspect the quality of identifiers in both, human-written manually and automatically generated, test code. As target systems, we select ten open source Java projects maintained by companies/organizations or by small teams of developers, ensuring high popularity and diversity of the target projects. The participants were asked to judge the identifiers and to list for the characteristics of high- and low-quality identifiers. To ease the interpretation of the achieved results and to have a baseline for comparison, we also asked four of the 19 participants to evaluate the quality of identifiers in the production code of two of the subject systems.

RQ_2: What is the accuracy of rename refactoring approaches when applied on test code identifiers? We evaluate three state-of-the-art rename refactoring approaches, namely CA-RENAMING [28], NATURALIZE [29], and LEAR [15]. We use the same ten projects used to answer RQ_1 and 429 additional projects from GitHub. We assess the rename refactorings with two different datasets as oracle: 1) the high-quality identifiers obtained as an output of RQ_1, 2) identifiers from the test code of open source projects that underwent code reviews. We also used the two systems for which we collected evaluations related to the quality of identifiers in production code to compare the performance of the renaming tools on the test and on the production code.

Our results show that low-quality identifiers are spread both in manually written and in automatically generated tests, and this problem is more relevant in test than in production code (RQ_1). State-of-the-art rename refactoring tools are of little help in improving the identifier quality of test code while their performance is more promising for production code (RQ_2). Major advances are needed in this field. Given our findings, we outline a research agenda for future work in the area.

II. Related Work

A. Quality of Identifiers

Strong connections have been discovered between bad identifier names and code quality issues [6]. Researchers have put a considerable amount of effort into investigating which characteristics of identifier names can influence program comprehension, positively or negatively.

Deissenboeck and Pizka [1] introduced two important concepts for good identifier naming: consistency and conciseness. They also proposed a model based on bijective mappings between concepts and names. The model requires that each concept should have a unique name and this name should be able to represent the concept correctly.

Lawrie *et al.* [2], [3] studied the impact of identifier length on program comprehension and found out that developers can easily comprehend source code with full word identifiers or well-formed abbreviations. However, excessively long identifiers might hinder program comprehension as they overload short-term memory. A recent study with 72 professional C# developers conducted by Hofmeister *et al.* [30] provides evidence that using full words in identifiers helps developers in code comprehension, compared to letters and abbreviations.

Lawrie *et al.* [31], [32] also analyzed identifier usage in 186 programs written in four different programming languages. Their findings disclose that better programming practices are producing higher quality identifiers.

Binkley *et al.* [11] conducted an experiment with 150 participants to understand the impact of identifier styles on program comprehension. As a result, they discovered that camel casing can help novices detect identifiers more accurately, at a cost of more time needed.

Researchers have also investigated practical issues (*e.g.,* bad smells, inconsistencies) originating from identifier naming. Kim *et al.* [33] performed interviews with developers, finding that developers often deal with inconsistent identifiers and the inconsistency is more common in larger projects. Butler *et al.* [34] analyzed 3.5 million Java reference name declarations in 60 well-known Java projects, and manually tagged around 46,000 names. Their study shows that the use of unknown abbreviations and words is not rare in the source code and might potentially hinder program comprehension.

Abebe *et al.* [35] introduced the notion of "lexicon bad smell" to indicate potential problems in identifier names. With the tool they built, they were able to identify 15,633 bad smells in *Alice*, an open-source software system containing around 1.5 million lines of code, demonstrating the wide spread of imperfect identifiers.

Arnaoudova *et al.* [36] presented a catalogue of 17 linguistic antipatterns (LAs) capturing inconsistencies among the naming, documentation, and implementation of attributes and methods, showing that LAs are negatively perceived by developers who highlighted their negative impact on code comprehension.

Fakhoury *et al.* investigated how poor lexica of source code negatively affects the readability of source code, thus hindering comprehension processes [37].

To the best of our knowledge, our study is the first focusing on the quality of identifiers used in **test code**.

B. Rename Refactoring

Identifiers are often composed of abbreviations, and techniques like identifier splitting [38]–[41] and expansion [42], [43] have been proposed to ease comprehension. However, in practice, lots of identifiers do not follow naming conventions and can be composed of meaningless tokens. Researchers have also investigated rename refactoring approaches, which rename the identifier with a more meaningful and/or consistent name.

Corbo *et al.* [44] and Reiss [45] proposed renaming approaches able to learn code identifier conventions from existing code. The rename refactoring approaches proposed by Feldthaus and Møller [46] and by Jablonski and Hou [47], instead, focus on the relations between variables, inferring whether one variable should be changed together with others.

Caprile and Tonella [13] proposed an approach to enhance the meaningfulness of identifiers with a standard lexicon dictionary and a thesaurus collected by analyzing a set of programs, replacing non-standard terms used in identifiers with a standard one from the dictionaries.

Thies and Roth [28] proposed a static analysis based approach to support identifier renaming: if a variable v_1 is assigned to an invocation of method m (*e.g.,* `name = getFullName`), and the type of v_1 is identical to the type of the variable v_2 returned by m, then rename v_1 to v_2. This was effective when experimented on open source projects.

Allamanis *et al.* [29] proposed NATURALIZE, a n-gram language model based approach which suggests new names to identifiers. The n-gram model predicts the probability of the next token given the previous n-1 tokens. NATURALIZE learns coding conventions from the codebase, promoting the consistent use of identifiers. The approach trains a language model on the rest of the project code, and then predicts the identifier names for the target files. Building on top of NATURALIZE, Lin *et al.* [15] proposed LEAR, an approach combining code analysis and n-gram language models. The differences between LEAR and NATURALIZE are 1) while NATURALIZE considers all the tokens in the source code, LEAR only focuses on tokens containing lexical information; 2) LEAR also considers the type information of variables.

The approach proposed by Daka *et al.* [48] is explicitly designed to rename identifier in test code and, in particular, in automatically generated unit tests. It generates descriptive method names for automatically generated unit tests by summarizing API-level coverage goals. A relevant work by Høst and Østvold [10] identifies the "bugs" in method names, meaning names that do not reflect the responsibilities implemented in the method. This approach recommends new method names by learning naming rules from a corpus of Java applications. Since these tools [10], [48] only recommend method names, they cannot be used in our study to suggest names for variables.

We assess the accuracy of three identifier renaming techniques [15], [28], [29] when applied on test code, including a comparison of their performance on production code.

III. Study I: Quality of Identifiers in Test Code

Our *goal* is to better understand the characteristics of good/bad identifiers used by local variables in test methods.

A. Research Question

Studies [2], [4], [10] have investigated the quality of identifiers in production code, yet little attention has been given to test code. We aims to answer the research question:

RQ_1: *What is the quality of identifiers in the test code of open source projects?*

The quality of the identifiers was judged by 19 participants, who were also required to justify their quality assessment by explicitly reporting what makes an identifier good or bad. Given the advances in automatic test case generation [27], we also asked participants to judge the quality of identifiers in automatically generated test cases for the same set of projects. Instructions were distrubuted to participants, stressing that high-quality identifiers make the code easier to read and understand.

The set of identifiers deemed as "good" in this study will be used as a ground truth in our second study (Section IV).This allows to have a manually validated ground truth, overcoming one of the limitations of experimentations performed to evaluate the performance of naming approaches, in which researchers often use the identifiers defined by developers in open source projects as oracle [14].

B. Study Context and Data Collection

TABLE I: Subject projects for Study I: Identifier quality.

Project	Repository	# Java Files	ELOC
Community Projects			
Commons Lang	https://goo.gl/wdZMf9	323	75,958
Gson	https://goo.gl/JkG9CV	176	22,272
Jackson Core	https://goo.gl/WTeh3N	238	42,150
Plexus-Utils	https://goo.gl/j3ckGk	128	24,710
REST Assured	https://goo.gl/ivx7jK	171	9,175
Team Projects			
Jesque	https://goo.gl/GJxAuv	121	10,339
Jongo	https://goo.gl/M2nDdK	155	8,190
la4j	https://goo.gl/fPKYDX	117	13,480
Natty	https://goo.gl/RBznPG	27	3,854
ORMLite Core	https://goo.gl/TXaRiR	280	34,970

The *study context* consists of the 10 open source Java projects from GitHub (Table I). We selected well-known projects maintained by companies/organizations (from now on *community projects*), as well as projects maintained by small teams (from now on *team projects*). We selected five projects for each of these two categories, by adopting the following selection criteria:

- **Popularity.** For *community projects*, we selected popular libraries hosted on Maven (https://mvnrepository.com/) and used by at least 500 client projects. For the *team projects*, we select projects having more than 300 stars on GitHub, to filter out "toy projects".
- **Diversity.** The projects are of different size and type and run by different entities, preventing the bias of internal coding conventions and programming practices.

To answer RQ_1, we conducted a survey asking 19 participants to manually inspect the quality of identifiers in both, human-written and automatically generated, test code.

Manually written test code. For each of the 10 projects, we randomly selected eight test methods from different classes to guarantee the generalizability and parsed them with JavaParser (https://javaparser.org/). In total, we extracted 237 manually written identifiers from these 80 test methods.

Automatically generated test code. We used EvoSuite [27] to generate test code for the selected projects, randomly selecting two test methods from each project. We collected 46 automatically generated identifiers.

Summarizing, we extracted 283 identifiers, 237 manually written and 46 automatically generated. We preferred to have more manually written than automatically generated identifiers since we expect automatically generated identifiers to follow a limited number of naming patterns and, thus, a smaller number of instances is necessary to observe a trend in the data.

We also asked four participants to judge the quality of identifiers in the production code of Jackson Core and ORMLite Core (*i.e.,* one community and one team project). This was done to (i) have a term of comparison when discussing the results achieved in terms of quality of the identifiers in test code, and (ii) verify whether there is a difference in the quality of identifiers used in test and production code. In this case, we extracted 47 identifiers from 20 methods (10 per system) contained in 20 different classes of the two systems. Note that the study on the production code identifiers has only been conducted on two systems since we preferred to polarize the participants' effort towards the evaluation of test identifiers, being this the main goal of our study.

Table II summarizes the identifiers judged for each project.

TABLE II: Number of identifiers inspected for each project

Project	#Human Written Test Identifiers	#Auto. Gener. Test Identifiers	#Human Written Prod. Identifiers	Total
Commons Lang	17	7	-	24
Gson	20	4	-	24
Jackson Core	38	8	26	46
Plexus-Utils	16	2	-	18
REST Assured	12	3	-	15
Jesque	25	7	-	32
Jongo	15	3	-	18
la4j	26	6	-	32
Natty	28	3	-	31
ORMLite Core	40	3	21	43
sum	237	46	47	330

1) Judgment of identifiers quality: Through convenience sampling, we invited 19 participants, including 4 professional developers, 11 computer science students (BSc, MSc, PhD), 2 academic staff to evaluate the quality of the identifiers collected from the previous steps, based on how well the identifiers support code comprehension. Participants had an average of 6.6 years experience of Java development (median=7.0, min=1, max=15), 1.5 years industrial experience (median=1, min=0, max=5), and 2.8 years experience of software testing (median=1, min=0, max=12). None of the participants was involved in the development of the subject projects.

TABLE III: Evaluation of identifier quality given by evaluators

Evaluation of Identifiers	Manually Written		Automatically Generated	
both good	59	(24.9%)	0	(0.0%)
both acceptable	13	(5.5%)	4	(8.7%)
both poor	36	(15.2%)	9	(19.6%)
both unsure	0	(0.0%)	0	(0.0%)
good & acceptable	43	(18.1%)	8	(17.4%)
good & poor	36	(15.2%)	6	(13.0%)
good & unsure	2	(0.8%)	0	(0.0%)
acceptable & poor	46	(19.4%)	19	(41.3%)
acceptable & unsure	1	(0.4%)	0	(0.0%)
poor & unsure	1	(0.4%)	0	(0.0%)
sum	237	(100.0%)	46	(100.0%)

Participants judged the identifiers from one test (or production) method at a time using a Web app we developed[1]. The app showed one test case/method at a time together with links to the methods in the production code that it tests. Participants are not explicitly informed whether the displayed method is manually written or automatically generated. The quality of an identifier was judged on a 3-point scale: "*good*", "*acceptable*", "*poor*". Participants could also select a "*not sure*" option.

Participants were asked to motivate their judgment by explaining the positive and negative characteristics of identifiers. An identifier judged as having a good (poor) quality could have both positive and negative characteristics. We provided two lists of predefined categories based on a literature review we performed (one for positive and one for negative characteristics, the detailed lists can be found in Section III-D), and participants could also add their own quality attributes. Moreover, they had the option to suggest a new name for the identifiers.

On average, each participant assessed the quality of 33.3 identifiers (median=23, min=16, max=118). Each identifier was evaluated by two participants, totaling 566 manual evaluations for test code and 94 for production code identifiers.

C. Data Analysis

To answer RQ_1, we plot the distribution of quality scores for the identifiers used in the subject test code. We discuss the characteristics of good and poor identifiers as reported by participants and compare the assessments provided for community projects and team projects, and the differences between human written and automatically generated identifiers. We also compare the quality of manually written identifiers in test and production code for Jackson Core and ORMLite Core.

D. Results

Table III reports the evaluations given by the participants to the quality of the identifiers subject of our study. Since each identifier has been judged by two evaluators, we report the frequency of each possible pair of evaluations and their ratio to the total number of evaluation pairs.

[1]The screenshots of the Web app can be found in the replication package: https://identifierquality.bitbucket.io/webapp/

TABLE IV: Frequency of scores given to identifier quality

Evaluation	Manually Written Identifiers		Automatically Generated Identifiers		Sum	
good	199	(42.0%)	14	(15.2%)	213	(37.6%)
acceptable	116	(24.5%)	35	(38.0%)	151	(26.7%)
poor	155	(32.7%)	43	(46.7%)	198	(35.0%)
unsure	4	(0.8%)	0	(0.0%)	4	(0.7%)
sum	474	(100.0%)	92	(100.0%)	566	(100.0%)

1) Agreement Analysis: Assessing the quality of an identifier is subjective and depends on the experience and coding habits of developers. We first look at the level of agreement reached by the study participants. For manually written variables, 45.6% of evaluations for the same identifier reached an agreement: both evaluators rate the same identifier as "*good*" (24.9%), "*acceptable*" (5.5%) or "*poor*" (15.2%). Since each identifier was judged on a 3-point scale, we also computed the cases of "weak agreement", meaning a 1-point difference on the quality assessment scale (*i.e.,* "*good vs acceptable*" and "*acceptable vs poor*"). In this case, the ratio of agreement reaches 83.1%. 15.2% quality assessments gave totally different quality scores (*i.e.,* "*good vs poor*"), which confirms that developers can have very different views on *what* a good identifier actually is.

For automatically generated variables, evaluators agreed in 28.3% of cases (as opposed to the 45.6% of the manually written code) and weakly agreed in 87.0% of cases. 13.0% obtained an inconsistent assessment (*i.e.,* "*good vs poor*").

The obtained agreement level confirmed the high subjectiveness of this task. It also highlighted a good level of agreement in discriminating between *good* and *poor* identifiers, with only ∼15% of identifiers falling in this strong disagreement scenario.

We also manually inspected these ∼15% of identifiers, and illustrate them with some examples. One interesting controversial identifier is "`notDao`". In that test case, "`dao`" was created to represent an object of type `LocalBigDecimalNumeric`. The developer used "`notDao`" to represent another object of a different numeric class. While one evaluator believes this identifier is informative, the other considers "`notDao`" as misleading as readers might consider it as a Boolean value. Another example is the identifier "`value`", assigned to a string "`easter '06`". "`value`" is intended to be parsed by a date parser. While one evaluator thinks this identifier is meaningful and concise, the other believes "`value`" is too general.

2) Quality of Identifiers: Table IV reports the quality scores assigned by participants to test code identifiers.

Manually written *vs* automatically generated. For manually written identifiers, 42% of the ratings indicate a *good* quality and an additional 24.5% an *acceptable* quality. ∼33% of evaluations pointed to *poor*-quality identifiers. This indicates that poor identifiers are frequent in manually written test code.

For automatically generated variables we obtained only 15.2% *good* evaluations (as compared to the 42% of manually written ones), with an additional 38% of *acceptable* ratings, *i.e.,* evaluators were not satisfied with the quality of identifiers in automatically generated test cases in almost half of the cases.

If we compare the results for manually written and automatically generated variables, the quality of manually written identifiers is better overall, especially considering that the automatic test case generation approach rarely generates "good" identifiers according to the study participants. It is worth highlighting that in ∼53% of the cases the evaluators considered the automatically generated identifiers at least as *acceptable*, indicating the use of good naming heuristics in EvoSuite.

We further analyze the obtained results in Section III-D3 to better understand the reasons behind these quantitative findings.

Community projects *vs* team projects. Table V reports the quality scores assigned to manually written variables in *community projects* and *team projects*.

TABLE V: Evaluation of manually written variables

Evaluation	Community Projects Manual Variables		Team Projects Manual Variables	
good	65	(31.6%)	134	(50.0%)
acceptable	62	(30.1%)	54	(20.1%)
poor	77	(37.4%)	78	(29.1%)
unsure	2	(1.0%)	2	(0.7%)
sum	206	(100%)	268	(100%)

A quality score is assigned by a single participant to one identifier (*i.e.,* a single evaluation). This means that each identifier results in two quality scores assigned by the two participants evaluating it, thus 237 manually written identifiers lead to 474 scores. As explained before, the same identifier could have both a *good* and a *poor* evaluation.

We can see from the table that for *community projects*, the ratios of "good", "acceptable", and "poor" quality evaluations are quite similar (∼30%), while for *team projects* around half of the evaluations pointed to a good identifier quality. This seems to indicate that the presence of organizations behind *community projects* does not guarantee better code quality assurance, at least not for identifiers quality in test code.

TABLE VI: Quality of Identifiers in Test Code *vs* Production Code for Jackson Core and ORMLite Core

Evaluation	Test Code Manual Variables		Production Code Manual Variables	
good	55	(35.3%)	49	(52.1%)
acceptable	41	(26.3%)	38	(40.4%)
poor	58	(37.2%)	7	(7.5%)
unsure	2	(0.2%)	0	(0.0%)
sum	156	(100%)	94	(100%)

Test Code *vs* Production Code. Finally, we conclude our quantitative analysis by comparing the quality of manually written identifiers in test and production code as judged by four participants for two subject systems (*i.e.,* Jackson Core and ORMLite Core). Table VI shows the achieved results: For production code, 92.5% of identifiers are judged as having a good or an acceptable quality, as compared to the 61.6% of the test code identifiers from the same systems. While a full comparison of the quality of identifiers in test and production code is out of the scope of this paper, the results obtained on these two systems seem to indicate that the quality problem is more evident in test code rather than in production code. Additional data is present in our online appendix [49].

3) Qualitative Analysis: Fig. 1 summarizes the reasons provided by participants when classifying a test code identifier as having a good quality (a), an acceptable quality (b), or a poor quality (c). These reasons are the characteristics that make an identifier perceived as good, acceptable, or poor. We did not report characteristics listed in less than 1% of cases.

Concerning "good" identifiers, "*it expresses a pattern*", "*it is too general*", and "*it uses a useless sequence number*" are the characteristics provided by the evaluators, while all others were predefined by us, based on the related literature. Among the listed characteristics, the most selected ones are "*it is meaningful*" and "*it is concise*": Participants appreciated short identifiers having, however, a clear meaning (*e.g.,* `config` is considered good as it refers to an object of type `HeaderConfig`).

Two factors considered by evaluators as contributing to high quality identifiers are semantic consistency (*i.e.,* no different identifiers are used for the same concept), and consistency with the tested code (*i.e.,* it uses the same terms used in the tested code to represent a specific concept). For example, the methods `setIndexName` and `getIndexName` appear in multiple test cases, all the identifiers they interact with are consistently named as `indexName`, without any use of other names such as `index` and `name`. Moreover, `indexName` is also consistently used in the methods tested by these test cases.

Good-quality identifiers also had some negative characteristics highlighted by the participants, and in particular "*it is too general*" (*e.g.,* when an object is named with the name of the class it instantiates) and "*it uses a useless sequence number*" which, as we will also discuss in the following, is one of the main issues with the automatically generated identifiers.

Moving to the poor-quality identifiers, besides the predefined characteristics, one additional characteristic has been contributed by the evaluators: "*it does not represent its type*". Fig. 1-(c) shows that different problems exist in the low-quality *manually written* and *automatically generated* variables. For *manually written variables*, the major issues include: 1) "*the identifiers are not meaningful*" (*e.g.,* `a` for a matrix); 2) "*the identifiers are too general*" (*e.g.,* `type` for the type of a token); and 3) "*the identifiers are too short*" (*e.g.,* `g` for a `JsonGenerator` object). Two other attributes which account for around 5% of occurrences each are "*syntactically similar to another identifier*" (*i.e.,* similar identifiers are used for other concepts, such as `applicationConfigurator` and `applicationConfiguration`) and "*not representing its type*" (*e.g.,* `strings` is used to name a `DateMap` object).

For *automatically generated variables*, the dominant issue is that identifiers include "*useless sequence numbers*". Indeed, EvoSuite assigns the object type as variable names followed by a progressive number (*e.g.,* a new instance of a `JsonReader` object is called `jsonReader0`). This heuristic, while very simple, helps EvoSuite obtain some meaningful identifiers, especially in the case where a single variable of a specific type is used (*e.g.,* a single `JsonReader` is instantiated).

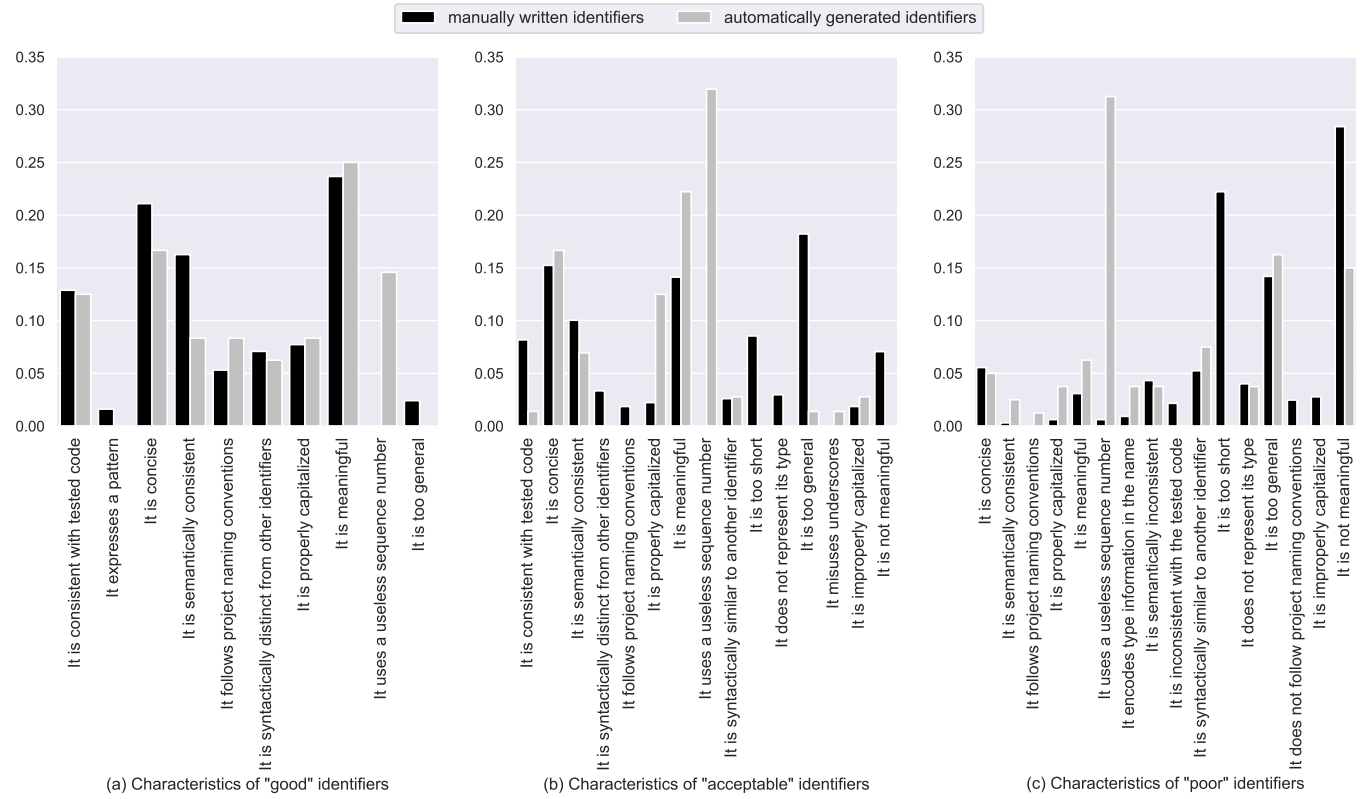


Fig. 1: Characteristics of identifiers having different quality levels, as perceived by the study participants

In this case, the progressive number is not disturbing and there is no reason for a more specific name (since only one variable of that type exists in the test method), explaining why the identifiers are assessed as *good* by the participants, despite the presence of a "*useless sequence number*" (see Fig. 1-(a)).

More specific names and advanced heuristics are needed when the role played in the test method by two variables of the same type must be *disambiguated* through their identifiers.

Being "*not meaningful*" and "*too general*" are two evident problems for automatically generated identifiers, accounting for 12% and 14% of the negative characteristics mentioned by the evaluators for the automatically generated tests. In very few cases, evaluators report the *misuse of underscore or of capitalization* as negative characteristics of identifiers in both *manually written* and *automatically generated* variables. These are issues that could be easily fixed with existing tools.

Finally, the *acceptable* identifiers (see Fig. 1-(b)) represent a mix of good and bad practices, justifying their rating in between good and poor identifiers.

4) Participants' recommendations to improve poor identifiers: As previously said, participants could suggest a new name for an identifier, when it was judged as not good enough. The recommendations can be found in our replication package. By inspecting the identifiers rated as *good* and the 205 identifiers suggested by participants, we observed three patterns:

① Participants prefer full name identifiers to abbreviations. For example, both evaluations judging the quality of the `qb` identifier recommended to rename it into `queryBuilder`, thus confirming the importance of techniques supporting the automatic expansion of identifiers (*e.g.,* [42], [43]).

② Plural format of an object type is recommended for the list of a certain type of objects. For example, `dataGroups` is suggested to replace `dataGroup`, which is a list of `DataGroup` objects.

③ Identifiers assigned to `get` methods and identifiers used as parameters of `set` methods are suggested to be consistent with the method names. For example, `foreignCollection` is considered a good name for a local variable assigned to the `getForeignCollection()` method.

We plan to conduct larger surveys in the future to distill a list of additional good naming practices and integrate them in rename refactoring and code generation tools.

IV. Study II: Identifier Renaming in Test Code

The *goal* of this study is to assess whether state-of-the-art rename refactoring techniques can improve the identifier quality, especially for the test code.

A. Research Question

Given the fact that the quality of identifiers in test code is indeed a problem, one might wonder whether we can automatically improve it. While rename refactoring techniques have been proven useful on the production code by several studies [15], [28], [29], their effectiveness on test code remains unknown.

We aim at answering the following research questions:

RQ$_2$: *What is the accuracy of rename refactoring approaches when applied on test code identifiers?*

This RQ aims at exploring the possibility of using state-of-the-art rename refactoring techniques [15], [28], [29] to improve identifier quality of test code.

B. Study Context

The *study context* consists of the same ten projects used in our first study and 429 additional projects mined from GitHub and used for the training/test of the refactoring techniques.

To select the tools, we first investigated which rename refactoring techniques can be applied to rename *variable* identifiers. This led to the identification of three state-of-the-art approaches, namely CA-RENAMING[2] [28], NATURALIZE [29], and LEAR [15]. These techniques are described in Section II. We used the original implementations provided by the authors of NATURALIZE and LEAR, and reimplemented CA-RENAMING.

We consider two types of ground truths to assess the accuracy of the experimented techniques. One is the set of 201 high-quality identifiers obtained as output of Study I, including the identifiers that were assessed by both evaluators as at least *acceptable* (*i.e.,* good-good, good-acceptable, acceptable-acceptable) as well as the identifiers suggested by participants as a good alternative to the poor identifiers. From now on, we refer to this ground truth as the *manual-oracle*. The second set includes reviewed test code identifiers used in the 429 additional projects we mined for this study (from now on, *mined-oracle*). Similarly to what has been done in the literature, the idea for the *mined-oracle* is to assess the ability of the experimented techniques in recommending identifiers for a given variable in a test method. The assumption is that these identifiers are meaningful and, as seen in RQ$_1$, such a strong assumption does not always hold, since low-quality identifiers are still prevalent in manually written code. We mitigate this issue in two ways. First, we also compute the accuracy of the rename refactoring techniques on the *manual-oracle* including manually checked identifiers assessed to be meaningful. Second, we only consider in *mined-oracle* identifiers from the test methods of the 429 projects that have been submitted in pull requests on GitHub and underwent a code review process. This should increase the confidence in the high quality of the identifiers in *mined-oracle*.

To understand how the performance of rename refactoring approaches differ for production code, we also constructed the *manual-oracle* for production code identifiers in the same way with the data collected in our first study, which consists of 42 identifiers from Jackson Core and ORMLite Core.

To build the *mined-oracle*, we first mined Java projects from GitHub on Sept. 1, 2018, using the following selection criteria:

- **Activity level.** To exclude inactive projects, the projects must have at least one commit in the three months preceding the data collection.
- **Popularity.** Projects must have at least 100 forks and 100 stars, in order to exclude "toy-projects".

[2]Note that CA-RENAMING is not the original name proposed by Thies and Roth, the researchers presenting this approach (that has no specific name), but the name assigned in [15], in which LEAR was compared to CA-RENAMING.

This process resulted in the selection of 2,583 Java projects. Then, we excluded the projects for which the test methods that underwent a review process in the latest version have less than 50 identifiers usable in our dataset, to ensure a good representativeness for each of the included projects. This led to the final 429 projects part of our dataset, including 24,355 reviewed test files. The test files were identified when their name started with "`Test`" or when they were located under a folder named "`src/test`" or "`tests`". Table VII summarizes the dataset used in this study.

TABLE VII: Dataset Statistics

	Overall	**Per Project**		
		Mean	Median	St. Deviation
Java files	166,558	388.2	256.0	385.6
# total test files	46,260	107.8	62.0	138.7
# test files for study	24,355	56.8	24.0	99.7
# variables for study	397,936	927.6	396.0	1533.6

C. Data Collection and Analysis

The three considered rename refactoring techniques rely on a training phase to learn naming patterns: LEAR [15] and NATURALIZE [29] need to build a language model based on n-grams extracted from the training code, while CA-RENAMING [28] needs to extract static type information and returned identifiers from declared methods in the training code. We experimented with different training scenarios to understand whether projects themselves or other projects are more helpful for training recommenders to rename identifiers in test code:

- **Training on production code.** Given the test code of a system A on which we apply a given renaming technique T, we train T on A's production code. Thus T learns naming conventions that are specific to project A.
- **Training on test code.** We train T on the large corpus of test code, extracted from reviewed and merged pull requests of 429 open source projects. Thus T learns naming conventions specific for test code, across several projects. The idea behind this scenario is that software is in general very repetitive and natural [50]. Due to the high computational cost of this procedure, this second training scenario has only been performed by using the 429 projects for training and the 10 projects used in Study I as testing (*i.e.,* the ones part of the *manual-oracle*).

We ran the three techniques on the *manual-oracle* (both scenarios) and on the *mined-oracle* (only in the "training on production code" scenario). For production code identifiers, we only performed training on other production code in the project. We also only ran the three techniques on the *manual-oracle*.

We used two different matching approaches to determine whether the techniques provide correct renaming recommendations: 1) exact match (the recommended identifier is identical to the one in the oracle); 2) fuzzy match, meaning that at least 50% of the tokens (words identified through CamelCase splitting) composing the identifier in the oracle appear among the tokens used in the recommended identifier.

For test code identifiers, we compare via box plots the precision of the techniques in the different training scenarios and on the two oracles. The comparisons are also performed via the Mann-Whitney test [51], with results intended as statistically significant at $\alpha = 0.05$. To control the impact of multiple pairwise comparisons (*e.g.,* the precision of CA-RENAMING is compared with both NATURALIZE and LEAR), we adjust p-values with the Holm's correction [52]. We estimate the magnitude of the differences by using the Cliff's Delta (d), a non-parametric effect size measure [53]. We follow well-established guidelines to interpret the effect size: negligible for $|d| < 0.10$, small for $0.10 \leq |d| < 0.33$, medium for $0.33 \leq |d| < 0.474$, and large for $|d| \geq 0.474$ [53].

For identifiers in production code, we list in a table the precision of the techniques for the two projects, and we also display the performance of the same projects when applying these techniques in their test code.

D. Results

1) Training on production code: We analyze the performance of rename refactoring techniques from two aspects: 1) the ability to generate recommendations for rename refactoring, 2) the correctness of the generated recommendations.

Ability to generate recommendations. Understanding how many recommendations can be generated can help us assess the applicability of rename refactoring tools in practice. Therefore, as the first step of our analysis, we inspect the percentage of identifiers involved in our study for which the three techniques can recommend an identifier name. Note that with "recommending an identifier name" we do not refer to the scenario in which a **new** name is recommended for a variable, but to the scenario in which a name (any name) is recommended, even the original one. Indeed, for a given variable, the three techniques might not be able to generate a recommendation. In particular, CA-RENAMING does not generate a recommendation in the case in which: 1) the variable to rename is not assigned to a method invocation (*e.g.,* for `String name = "Max"`, CA-RENAMING cannot be applied — see Section II for a description of the CA-RENAMING technique) or if the invoked method returns a variable of a different type (*e.g.,* `String age = (String) getAge()` with `getAge()` returning an integer). The other two techniques (NATURALIZE and LEAR) are both based on n-gram language models, and do not trigger any recommendation when a minimum confidence threshold set by the original authors is not met for a generated identifier.

Tables VIII and IX report descriptive statistics (*e.g.,* mean across projects) of the ratio of variables with renaming recommendations for test code generated by CA-RENAMING, NATURALIZE and LEAR on the *manual-oracle* and the *mined-oracle*, respectively. CA-RENAMING is omitted in Table VIII as it is unable to generate any renaming recommendation.

The achieved results show the limited percentage of cases in which these approaches are actually able to generate a recommendation. Indeed, even by considering the approach generating the highest number of recommendation (*i.e.,* LEAR), it can only be applied on ∼20% of the test code identifiers of a given project. Not surprisingly, CA-RENAMING has the lowest applicability, given its strong constraint making it applicable only to variables assigned to a method invocation returning the same type. NATURALIZE can generate refactoring recommendations for around 10% of the variables in the *manual-oracle*, while for *mined-oracle* this percentage significantly drops. This difference might be the consequence of test method sampling when building the *manual-oracle* dataset.

TABLE VIII: Ratio of variables for which a rename refactoring is generated (*manual-oracle*)

Approach	Mean	Median	St. Deviation
NATURALIZE	12.2%	9.9%	0.133
LEAR	22.1%	17.9%	0.199

TABLE IX: Ratio of variables for which a rename refactoring is generated (*mined-oracle*)

Approach	Mean	Median	St. Deviation
CA-RENAMING	0.9%	0.1%	0.023
NATURALIZE	3.6%	0.0%	0.075
LEAR	26.1%	24.0%	0.251

Correctness of the generated recommendations. Fig. 2 compares the precision of rename refactoring techniques when applied on the *manual-oracle* and the *mined-oracle*.

For the reason mentioned before, since CA-RENAMING does not generate any recommendation for the *manual-oracle*, it is not plotted on Fig. 2a. The main message highlighted by Fig. 2 is that the precision is in general quite low in terms of recommending good identifiers for test code. Moreover, although LEAR significantly outperforms the other two approaches (see Table XI), the average and median precision is still lower than 50% even when only fuzzy match is required.

However, it is worth noting that the low precision does not necessarily mean the generated identifiers are wrong, due to the matching rules we adopted to define "correctness". As we know, in practice, often many variants of identifiers can well fit in the code context. Therefore, the precision people perceive with these tools could be higher than the values presented here.

To better compare these rename refactoring approaches, we applied statistical analysis to the precisions of the renaming recommendations. For the *manual-oracle*, we compared NATURALIZE against LEAR. The p-value of 0.35 (exact match)/0.44 (fuzzy match) indicates that the precision difference between NATURALIZE and LEAR is not statistically significant. However, the situation changes on the *mined-oracle*. In the Table XI, we can find that there is no statistically significant difference (adjusted p-value ≥ 0.05) between CA-RENAMING and NATURALIZE when exact match is required. However, the advantage of LEAR is visible in any case. All of the statistical comparisons with CA-RENAMING and NATURALIZE result in a statistically significant difference, with small or negligible effect sizes.

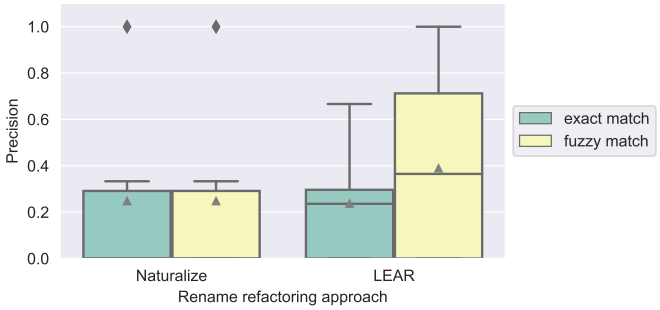
Test Code vs Production Code. Table X compares the performance of rename refactoring techniques when they are applied to production code and test code (*manual-oracle*).

TABLE X: Performance comparison of rename refactoring techniques for identifiers in production code and test code

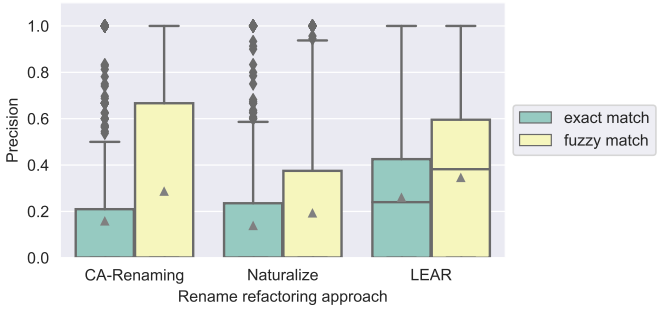
Project	# Variables	NATURALIZE # Recomm.	NATURALIZE Precision (exact)	NATURALIZE Precision (fuzzy)	LEAR # Recomm.	LEAR Precision (exact)	LEAR Precision (fuzzy)
Jackson Core (Prod. code)	24	5	40.0%	40.0%	20	55.0%	60.0%
Jackson Core (Test code)	35	4	0.0%	0.0%	3	0.0%	0.0%
ORMLite Core (Prod. code)	18	8	37.5%	62.5%	7	42.9%	57.1%
ORMLite Core (Test code)	37	0	0.0%	0.0%	0	0.0%	0.0%

TABLE XI: Statistical tests of precisions of rename refactoring techniques for *mined-oracle*

Comparison	P-Value (exact match)	P-Value (fuzzy match)	Effect Size (exact match)	Effect Size (fuzzy match)
CA-RENAMING *vs* NATURALIZE	0.74	0.0037	0.01 (Negligible)	0.10 (Negligible)
CA-RENAMING *vs* LEAR	<0.0001	0.0003	0.29 (Small)	0.14 (Negligible)
NATURALIZE *vs* LEAR	<0.0001	<0.0001	0.31 (Small)	0.29 (Small)



(a) Precision of rename refactoring techniques on *manual-oracle*



(b) Precision of rename refactoring techniques on *mined-oracle*

Fig. 2: Precision of rename refactoring techniques on test code

CA-RENAMING is also omitted as no recommendation was generated for both production and test code. We can notice that rename refactoring approaches can generate more recommendations for production code, and the precision is much higher. This result indicates that rename refactoring techniques are less effective when used to improve the quality of test code identifiers as compared to production code identifiers.

2) Training on test code: Table XII reports the performance of NATURALIZE and LEAR applied on the *manual-oracle* of test code identifiers, when training on test code from other projects.

In this case, CA-RENAMING was unable to generate any recommendation, as it heavily relies on program analysis. Since no production code was used for training, CA-RENAMING could not retrieve the declarations of methods used in test cases. Therefore, CA-RENAMING is excluded in this study.

Both NATURALIZE and LEAR perform poorly in this task. The unsatisfactory performance comes from two aspects: the amount and the precision of generated refactoring recommendations. More specifically, NATURALIZE failed to generate recommendations for six projects, while LEAR could not recommend any identifier for five projects. As a side note, LEAR can generate at maximum five refactoring recommendations when applied on the *manual-oracle* and trained with *test code*. When it comes to the precision of exactly matched recommendations, the performance is extremely poor for LEAR. That is, none of the generated recommendations is correct, which is not the case for NATURALIZE.

We can also spot some major differences between these results and the previous ones. Although the performance of both techniques drop significantly, in this study NATURALIZE performs better than LEAR in terms of the number of exactly matched generated recommendations. The reason could be the nature of the training materials. Unlike the previous study, in which the training of the techniques was performed on the production code of the same system for which the test code identifiers were recommended, training on the test code from other projects likely results in the learning of linguistic patterns that are not representative of the "test project" (*i.e.,* the one for which identifiers must be recommended). This might be due to a vocabulary mismatch between the code used for training and the one used for test. LEAR seems to be more sensitive to this change since it only considers tokens carrying out semantic information during the training (*i.e.,* the identifiers used in method names, parameters, and variables), while NATURALIZE, also learns from syntax-related tokens (*e.g.,* Java keywords), thus being able to better deal with the vocabulary mismatch.

Although researchers have proved that source code is repetitive [50], [54], [55], our study discloses that to recommend renaming operations for test code, it might be more effective to train these approaches on the related production code rather than from a massive dataset containing thousands of projects.

TABLE XII: Results of rename refactoring techniques for *manual-oracle* when trained on test code

Project	# Variables	NATURALIZE # Recomm.	NATURALIZE Precision (exact)	NATURALIZE Precision (fuzzy)	LEAR # Recomm.	LEAR Precision (exact)	LEAR Precision (fuzzy)
Natty	22	12	50.0%	50.0%	1	0.0%	0.0%
Jongo	12	0	0.0%	0.0%	0	0.0%	0.0%
Commons Lang	14	10	0.0%	0.0%	1	0.0%	0.0%
Jackson Core	35	0	0.0%	0.0%	3	0.0%	66.7%
Plexus-Utils	14	0	0.0%	0.0%	5	0.0%	40.0%
Jesque	21	10	10.0%	10.0%	0	0.0%	0.0%
Gson	18	14	28.6%	43.0%	0	0.0%	0.0%
REST Assured	12	0	0.0%	0.0%	0	0.0%	0.0%
la4j	16	0	0.0%	0.0%	0	0.0%	0.0%
ORMLite Core	37	0	0.0%	0.0%	2	0.0%	50.0%

V. Threats to Validity

Construct validity. In Study I, instead of using proxy measures, we preferred to let participants evaluate the quality of identifiers used in test code. While how to perceive the identifier quality may vary among different participants, the subjectiveness of such an evaluation was mitigated by involving two evaluators for each identifier. Also, although a four or five-level Likert scale [56] could have provided a more accurate evaluation of the identifiers' quality, we preferred a simpler three-level scale to facilitate the task to the respondents.

In Study II, we assessed the performance of the experimented techniques by adopting two different ground truths that complement each other. Indeed, the *manual-oracle* is small in size, but includes identifiers manually classified as meaningful. The *mined-oracle*, instead, includes 397,936 identifiers, thus ensuring a good generalizability at the risk, however, of including some poor-quality identifiers in the ground truth. This threat was mitigated by only considering in the *mined-oracle* identifiers from test code that underwent code review.

Internal validity. The experience of the participants involved in Study I could have played a role in the assessment of the identifiers quality. We only involved participants having at least one year of Java experience but, due to the limited number of participants, we did not analyze the influence of their experience on the quality assessments they provided.

External validity. The validity of Study I is limited by the 19 participants and by the selected projects of our study. This, as a consequence, partially impacted the generalizability of Study II concerning the results achieved on the *manual-oracle*. Also, when running our studies on production code, we only considered identifiers from two systems (and their respective evaluations provided by four participants in Study I). This is a clear limitation to the generalizability of the findings related to the comparison between test and production code performed in both studies. However, our focus is on test code identifiers, and production code identifiers were only considered to have a baseline for comparison, easing the interpretation of the achieved results. Details about the results achieved on production code identifiers are available in our appendix [49].

VI. Conclusion and Future Work

We studied the the quality of identifiers in test code and compared it with identifiers in production code. We also analyzed the attributes that are important for identifier quality and assessed the performance of three state-of-the-art rename refactoring techniques in suggesting good identifiers. The results of our study provide us with a few lessons learned.

The quality of identifiers in test code is a notable problem. Even in well-known projects run by open source organizations, one out of three quality assessments performed by developers would result in the identification of a poor-quality identifier. This highlights the need for techniques and tools able to help developers in identifying and fixing these problematic identifiers, and leads us to our next point.

The performance of state-of-the-art rename refactoring techniques is far from promising for improving the unsatisfactory identifier quality of test code. In the best case scenario, these techniques achieve a limited precision, lower than 50% on average. We observed that training language models on the production code of the same system for which test code identifiers should be recommended as a more promising training approach as compared to the usage of a large set of test cases extracted from other systems. Techniques specifically tailored for test code and, for example, exploiting its relationship with the tested production code, might be required to substantially increase the automated support provided to developers for the renaming of test code identifiers.

Automatically generated test code suffers even more from identifiers' quality issues. This result, while expected, highlights the need for integrating more sophisticated naming heuristics in tools for the automatic generation of test cases. Our findings in Study I disclose that some simple heuristics (*e.g.,* the use of plural for naming variables representing collections of objects) could be implemented with very little effort, and would generate identifiers appreciated by software developers.

These findings dictate our future research agenda.

Reproducibility. The data used in our studies as well as the experimented renaming approaches are available for replication (https://identifierquality.bitbucket.io/). This includes the *manual-oracle* output of Study I that could represent a valuable resource for testing rename refactoring approaches tailored for test code.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the projects PROBE (SNF Project No. 172799) and CCQR (SNF Project No. 175513).

REFERENCES

[1] F. Deissenboeck and M. Pizka, "Concise and consistent naming," *Software Quality Journal*, vol. 14, no. 3, pp. 261–282, 2006.

[2] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "What's in a name? a study of identifiers," in *Proceedings of ICPC 2006 (14th IEEE International Conference on Program Comprehension)*. IEEE, 2006, pp. 3–12.

[3] ——, "Effective identifier names for comprehension and memory," *Innovations in Systems and Software Engineering*, vol. 3, no. 4, pp. 303–318, 2007.

[4] E. W. Høst and B. M. Østvold, "Software language engineering," D. Gašević, R. Lämmel, and E. Wyk, Eds. Berlin, Heidelberg: Springer-Verlag, 2009, ch. The Java Programmer's Phrase Book, pp. 322–341.

[5] S. Butler, "The effect of identifier naming on source code readability and quality," in *Proceedings of ESEC/FSE 2009 Doctoral Symposium (Joint 12th European Software Engineering Conference and 17th ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM, 2009, pp. 33–34.

[6] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Relating identifier naming flaws and code quality: An empirical study," in *Proceedings of WCRE 2009 (16th Working Conference on Reverse Engineering)*. IEEE, 2009, pp. 31–35.

[7] L. Pollock, K. Vijay-Shanker, E. Hill, G. Sridhara, and D. Shepherd, "Natural language-based software analyses and tools for software maintenance," in *Software Engineering: International Summer Schools, ISSSE 2009-2011, Salerno, Italy. Revised Tutorial Lectures*. Springer, 2009, pp. 94–125.

[8] 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 *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*. IEEE, 2013, pp. 3–12.

[9] "Java language specification. https://docs.oracle.com/javase/specs/jls/se7/html/jls-6.html."

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

[11] D. Binkley, M. Davis, D. Lawrie, J. I. Maletic, C. Morrell, and B. Sharif, "The impact of identifier style on effort and comprehension," *Empirical Software Engineering*, vol. 18, no. 2, pp. 219–276, 2013.

[12] D. Binkley, M. Hearn, and D. Lawrie, "Improving identifier informativeness using part of speech information," in *Proceedings of MSR 2011 (8th Working Conference on Mining Software Repositories)*. ACM, 2011, pp. 203–206.

[13] B. Caprile and P. Tonella, "Restructuring program identifier names," in *Proceedings of ICSM 2000 (2000 International Conference on Software Maintenance)*, 2000, pp. 97–107.

[14] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names," in *Proceedings of FSE 2015 (10th Joint Meeting on Foundations of Software Engineering)*. ACM, 2015, pp. 38–49.

[15] B. Lin, S. Scalabrino, A. Mocci, R. Oliveto, G. Bavota, and M. Lanza, "Investigating the use of code analysis and NLP to promote a consistent usage of identifiers," in *Proceedings of SCAM 2017 (17th IEEE International Working Conference on Source Code Analysis and Manipulation)*, ser. SCAM '17, 2017, pp. 81–90.

[16] M. Beller, G. Georgios, A. Panichella, S. Proksch, S. Amann, and A. Zaidman, "Developer testing in the ide: Patterns, beliefs, and behavior," *IEEE Transactions on Software Engineering*, pp. 1–1, 2018.

[17] M. Beller, G. Gousios, A. Panichella, and A. Zaidman, "When, how, and why developers (do not) test in their ides," in *Proceedings of ESEC/FSE 2015 (10th Joint Meeting on Foundations of Software Engineering)*. ACM, 2015, pp. 179–190.

[18] A. Zaidman, B. Rompaey, A. Deursen, and S. Demeyer, "Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining," *Empirical Softw. Engg.*, vol. 16, no. 3, pp. 325–364, 2011.

[19] D. Athanasiou, A. Nugroho, J. Visser, and A. Zaidman, "Test code quality and its relation to issue handling performance," *IEEE Transactions on Software Engineering*, vol. 40, no. 11, pp. 1100–1125, Nov 2014.

[20] S. Counsell, G. Destefanis, X. Liu, S. Eldh, A. Ermedahl, and K. Andersson, "Comparing test and production code quality in a large commercial multicore system," in *Proceedings of SEAA 2016 (42th Euromicro Conference on Software Engineering and Advanced Applications)*, Aug 2016, pp. 86–91.

[21] F. Palomba, A. Zaidman, M. Bruntink, and A. Bacchelli, "On the relation of test smells to software code quality," in *Proceedings of ICSME 2018 (34th International Conference on Software Maintenance and Evolution)*, 2018.

[22] A. Deursen, L. M. Moonen, A. Bergh, and G. Kok, "Refactoring test code," Amsterdam, The Netherlands, The Netherlands, Tech. Rep., 2001.

[23] G. Bavota, A. Qusef, R. Oliveto, A. Lucia, and D. Binkley, "Are test smells really harmful? an empirical study," *Empirical Softw. Engg.*, vol. 20, no. 4, pp. 1052–1094, Aug. 2015.

[24] M. Tufano, F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, A. D. Lucia, and D. Poshyvanyk, "An empirical investigation into the nature of test smells," in *Proceedings of ASE 2016 (31st IEEE/ACM International Conference on Automated Software Engineering)*, Sept 2016, pp. 4–15.

[25] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia, "Automatic test case generation: What if test code quality matters?" in *Proceedings of ISSTA 2016 (25th International Symposium on Software Testing and Analysis)*. ACM, 2016, pp. 130–141.

[26] G. Grano, S. Scalabrino, H. C. Gall, and R. Oliveto, "An empirical investigation on the readability of manual and generated test cases," in *Proceedings of the 26th Conference on Program Comprehension, ICPC*, 2018, pp. 348–351.

[27] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of ESEC/FSE 2011 (19th ACM SIGSOFT Symposium on the Foundations of Software Engineering and 13th European Software Engineering Conference)*. ACM, 2011, pp. 416–419.

[28] A. Thies and C. Roth, "Recommending rename refactorings," in *Proceedings of RSSE 2010 (2nd International Workshop on Recommendation Systems for Software Engineering)*. ACM, 2010, pp. 1–5.

[29] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," in *Proceedings of FSE 2014 (22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering)*, ser. FSE 2014, 2014, pp. 281–293.

[30] J. Hofmeister, J. Siegmund, and D. V. Holt, "Shorter identifier names take longer to comprehend," in *Proceedings of SANER 2017 (24th International Conference on Software Analysis, Evolution and Reengineering)*. IEEE, 2017, pp. 217–227.

[31] D. Lawrie, H. Feild, and D. Binkley, "An empirical study of rules for well-formed identifiers," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 19, no. 4, pp. 205–229, 2007.

[32] ——, "Quantifying identifier quality: an analysis of trends," *Empirical Software Engineering*, vol. 12, no. 4, pp. 359–388, 2007.

[33] S. Kim and D. Kim, "Automatic identifier inconsistency detection using code dictionary," *Empirical Software Engineering*, vol. 21, no. 2, pp. 565–604, 2016.

[34] S. Butler, M. Wermelinger, and Y. Yu, "A survey of the forms of Java reference names," in *Proceedings of ICPC 2015 (23rd International Conference on Program Comprehension)*. IEEE, 2015, pp. 196–206.

[35] S. L. Abebe, S. Haiduc, P. Tonella, and A. Marcus, "Lexicon bad smells in software," in *Proceedings of SANER WCRE 2009 (16th Working Conference on Reverse Engineering)*. IEEE, 2009, pp. 95–99.

[36] V. Arnaoudova, M. Di Penta, and G. Antoniol, "Linguistic antipatterns: What they are and how developers perceive them," *Empirical Software Engineering*, vol. 21, no. 1, pp. 104–158, 2016.

[37] S. Fakhoury, Y. Ma, V. Arnaoudova, and O. O. Adesope, "The effect of poor source code lexicon and readability on developers' cognitive load," in *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018*, 2018, pp. 286–296. [Online]. Available: http://doi.acm.org/10.1145/3196321.3196347

[38] E. Hill, D. Binkley, D. Lawrie, L. Pollock, and K. Vijay-Shanker, "An empirical study of identifier splitting techniques," *Empirical Software Engineering*, vol. 19, no. 6, pp. 1754–1780, 2014.

[39] L. Guerrouj, M. Di Penta, G. Antoniol, and Y.-G. Guéhéneuc, "Tidier: an identifier splitting approach using speech recognition techniques," *Journal of Software: Evolution and Process*, vol. 25, no. 6, pp. 575–599, 2013.

[40] A. Corazza, S. Di Martino, and V. Maggio, "LINSEN: An efficient approach to split identifiers and expand abbreviations," in *Proceedings of ICSM 2012 (28th International Conference on Software Maintenance)*. IEEE, 2012, pp. 233–242.

[41] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker, "Mining source code to automatically split identifiers for software analysis," in *Proceedings*

of MSR 2009 (6th IEEE International Working Conference on Mining Software Repositories). IEEE, 2009, pp. 71–80.

[42] E. Hill, Z. P. Fry, H. Boyd, G. Sridhara, Y. Novikova, L. Pollock, and K. Vijay-Shanker, "AMAP: automatically mining abbreviation expansions in programs to enhance software maintenance tools," in *Proceedings of MSR 2008 (5th IEEE International Working Conference on Mining Software Repositories)*. ACM, 2008, pp. 79–88.

[43] D. Lawrie and D. Binkley, "Expanding identifiers to normalize source code vocabulary," in *Proceedings of ICSM 2011 (27th International Conference on Software Maintenance)*. IEEE, 2011, pp. 113–122.

[44] F. Corbo, C. Del Grosso, and M. Di Penta, "Smart Formatter: Learning coding style from existing source code," in *Proceedings of ICSM 2007 (23rd IEEE International Conference on Software Maintenance)*. Paris, France: IEEE CS Press, Oct. 2007, pp. 525–526.

[45] S. P. Reiss, "Automatic code stylizing," in *Proceedings of ASE 2007 (22nd IEEE/ACM International Conference on Automated Software Engineering)*. Atlanta, Georgia, USA: ACM Press, 2007, pp. 74–83.

[46] A. Feldthaus and A. Møller, "Semi-automatic rename refactoring for JavaScript," in *ACM SIGPLAN Notices*, vol. 48, no. 10. ACM, 2013, pp. 323–338.

[47] P. Jablonski and D. Hou, "CReN: a tool for tracking copy-and-paste code clones and renaming identifiers consistently in the IDE," in *Proceedings of ETX 2007 (2007 OOPSLA workshop on eclipse technology eXchange)*. ACM, 2007, pp. 16–20.

[48] 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 ISSTA 2017 (26th ACM SIGSOFT International Symposium on Software Testing and Analysis)*. ACM, 2017, pp. 57–67.

[49] "Replication package. https://identifierquality.bitbucket.io/."

[50] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Proceedings of ICSE 2012 (34th International Conference on Software Engineering)*. IEEE, 2012, pp. 837–847.

[51] W. J. Conover, "Practical nonparametric statistics," 1999.

[52] S. Holm, "A simple sequentially rejective multiple test procedure," *Scandinavian journal of statistics*, pp. 65–70, 1979.

[53] R. J. Grissom and J. J. Kim, "Effect sizes for research: A broad practical approach," *Mahwah, NJ: Earlbaum*, 2005.

[54] M. Gabel and Z. Su, "A study of the uniqueness of source code," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 2010, pp. 147–156.

[55] B. Lin, L. Ponzanelli, A. Mocci, G. Bavota, and M. Lanza, "On the uniqueness of code redundancies," in *Proceedings of ICPC 2017 (25th International Conference on Program Comprehension)*. IEEE, 2017.

[56] A. N. Oppenheim, *Questionnaire Design, Interviewing and Attitude Measurement*. Pinter Publishers, 1992.