The Journal of Systems & Software 205 (2023) 111821

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

A uniqueness-based approach to provide descriptive JUnit test names ☆

Jianwei Wu *, James Clause

Department of Computer and Information Sciences, University of Delaware, Newark, DE, USA

ARTICLE INFO

Article history:
Received 20 February 2023
Received in revised form 14 July 2023
Accepted 3 August 2023
Available online 8 August 2023

Keywords:
Test maintenance
Software testing
Documentation

ABSTRACT

The descriptive naming of unit tests has always been a focal task in test maintenance. Previous work tried to use different methods to generate descriptive names for unit tests or provide suggestions to improve existing names, but they often neglected developers' needs. Therefore, they are unlikely to be useful to provide descriptive test names for developers in real world. Based on a recent study that can identify uniqueness of JUnit tests, we propose a uniqueness-based name generation approach to generate descriptive test names that meet developers' needs. Comparing with several alternative approaches, the generated name from our approach are preferred by professional developers, or at least at the same level of preference as the original test names.

1. Introduction

Understanding the purpose of tests is the first step towards modifying existing tests, adding new tests, or removing unnecessary tests. Because the necessary information is often implicitly stated in the source code of unit tests and the classes under test, understanding this information is often the most difficult task in test maintenance. In order to avoid reading tens or hundreds of lines of code, summary information, written in natural language, is often provided in the form of a descriptive test name. If every test in a test suite has a descriptive name, a developer can find a test and understand its purpose quickly by only reading its name. The collection of test names can also serve as a specialized documentation for a test suite. Therefore, providing descriptive test names can significantly improve the efficiency of test maintenance tasks.

Unfortunately, existing work has shown that tests often do not have descriptive names (Zhang et al., 2016). Poor names are often written by developers and widely exist in the test suites of real world projects. For example, developers often create test names for convenience (e.g., testBug1, testB, etc.) and ignore future test maintenance needs (Zhang et al., 2016; Daka et al., 2017; Wu and Clause, 2020, 2022a). In addition, during test maintenance, developers might change the source code or context of tests without updating corresponding names (Wu and Clause, 2022a), and it would make existing test names erroneous.

There are two well-studied directions for solving the problem of poor names. For one direction, existing approaches use predefined rules (e.g., Zhang et al., 2016; Høst and Østvold, 2009; Arcuri et al., 2014), API-level coverage goals, or language models (e.g., Daka et al., 2017; Allamanis et al., 2015) to suggest better names. For the other direction, existing approaches attempt to solve the naming problem by automatically generating unit tests (i.e., both name and body) (Fraser and Arcuri, 2011; Zhang et al., 2015; Arcuri and Fraser, 2016; Jensen et al., 2013). While both directions of approaches can be successful at suggesting improvements to poor names or generating descriptive names and test cases, such names often do not meet with developer approval (e.g., because the generated names do not follow the developer's preferred naming rationales Wu and Clause, 2022a). More recent work attempts to generate names that more closely match existing names by using machine learning to create approaches that can map method bodies to method names (e.g., Alon et al., 2018, 2019). However, the success of these approaches heavily depend on their training sets and validation methods, so they are not always successful at creating models that match well with human cognition (Lison, 2015). In addition, these approaches are mainly built towards general methods and have problems in the descriptiveness of their generated names. For example, we have found that, due to the similarity of test bodies, these state-of-the-art approaches (i.e., Alon et al., 2018, 2019) often generate duplicate names for tests in the same class (see Wu and Clause, 2022a). While such names may make individual sense, they are useless in real world as duplicate names are not possible and, even if they were, they would not serve the goal of helping developers comprehend the purposes of their tests. Besides the usefulness in practice, existing work often invites students or random contributors from Github to their evaluation. Recent work has already shown that they cannot represent how professional developers would name a test (Beller et al., 2015). Therefore, the evaluation of existing work might be compromised by that, further losing support and becoming less successful in real world.

☆ Editor: Aldeida Aleti.
* Corresponding author.
E-mail addresses: wjwcis@udel.edu (J. Wu), clause@udel.edu (J. Clause).

https://doi.org/10.1016/j.jss.2023.111821
0164-1212/

As discussed above, existing approaches are limited in real world and cannot provide descriptive names that meet developers' approval. Therefore, further work is required in order to generate test names that are both descriptive, take into account the differences between test names and general method names, and meet with developer approval. Our work is inspired by a belief, learnt from our experiences with many tests from many projects, that a test's name is often based on what makes the test unique from its siblings—tests declared in the same class. This belief is confirmed with our previous work that focuses on automatically identifying the uniqueness of unit tests (Wu and Clause, 2022a). The results of the previous work: (1) confirm our impression that the majority of tests are named after what makes them unique, and (2) help us identify additional aspects that influence how tests are named.

This paper is a direct successor of our previous work (Wu and Clause, 2022a). Based on the identified attributes from the previous work and a set of attribute transformations in this paper, we propose a uniqueness-based approach to provide descriptive test names with developers' approval. To evaluate our approach, we considered 3 alternative approaches to compare our approach with. Using the generated names from our approach and alternatives, we performed an empirical survey with 5 professional developers from top-tier technology companies as participants to impartially judge the quality of the generated names from each considered approach. Moreover, our approach was tuned in the worst case setting in the evaluation. Even with the worst case setting, the generated names from our approach are similar to the original names and outperform other alternative approaches judged by the participants.

More specifically, this paper makes the following contributions:

(1) a uniqueness-based approach that can provide descriptive test names
(2) an empirical evaluation that demonstrates, among several alternatives, participants often agreed with the generated names from our approach similar to the original names, and our approach earned a significantly higher level of agreement than the alternatives.

The rest of this paper is structured as follows. Section 2 introduces our uniqueness-based naming approach and its related data analysis, Section 3 describes the details of the empirical evaluation with a statistical comparison between our approach and alternatives, Section 4 describes several possible threats to our current work and how to address them, Section 5 describe the related work of this paper, and Section 6 describes the conclusion of this paper.

2. Uniqueness-based unit test naming approach

As the basis for our name generation approach, we are using our recent work on automatically identifying the unique aspects of a test (Wu and Clause, 2022a). That work was motivated by a large-scale empirical study of the tests from 11 projects that demonstrates that the unique aspects of a test are often the basis for its name. Based on the results from the empirical study, we designed a novel, automated approach that can extract the attributes of a test that make it unique among its siblings. At a high-level, the approach uses a combination of static program analysis—to extract a variety of candidate attributes from a test suite—and formal concept analysis—to identify which combination of attributes is unique to the test under consideration.

For example, Fig. 1 shows a test and the output of the uniqueness-identification approach. The identified attributes of the test are: `putAsync` and `getAsync` (both are calls to methods defined by the class under test). Those attributes are unique because no sibling (i.e., tests in the same class) contains both method calls. While the output of the approach agrees with human judgement about what features of a test make it unique, they cannot be directly used as a test name.

In order to conform with expected naming conventions, the unique attributes of a test must be transformed and merged in various ways. For the example from Fig. 1, the test name `TestCacheAsync` shows the result of this process. This name was created, based on the unique attributes, by human participant involved the in evaluation of the uniqueness-identification approach. In this case, it appears that the participant used three transformations to construct the name from the unique attributes. First, the existing attributes are merged into a single attribute, `Async`, by removing the leading `put` and `get`. Second, an additional attribute—`Cache`, a variable name of the class under test—is added to the set of attributes. Third, the attributes are ordered, in this case, based on execution order. Finally, each word in the attributes are capitalized and the resulting attributes are joined together, in order, with a leading `Test`. In order to create a successful approach for generating test names based on unique attributes, it is necessary to understand the types of modifications and transformations that would be performed by a human in more details. The next section describes our processing for investigating this question.

Table 1
Considered projects.

Project	Version	LoC	# Tests
Sentinel	10c92e6	79,385	488
Jedis	d7aba13	36,582	684
Jfreechart	4e0a53e	133,540	2,176
Redisson	6feb33c	150,703	2,036
Spark	7551a7d	11,956	310
Webmagic	96ebe60	15,774	98
Picasso	a087d26	11,006	229
Fastjson	e05f1f9	195,511	4,950
Moshi	dbed99d	22,168	716

2.1. Identifying how to transform the unique aspects of a test into a name

In order to understand the types of transformations that are often made to create a descriptive test name, we used data collected from the evaluation of the uniqueness-identification approach (Wu and Clause, 2022a). This evaluation asked 3 participants about the performance of the uniqueness-identification approach when applied to 45 tests randomly selected from 9 top-ranked GitHub Projects (i.e., 5 tests from each project). The considered projects can be found in Table 1. As part of the evaluation, the participants were asked to consider a test body and the unique attributes that were identified by the approach. Then they were asked, if they were to create a name for the test based on the attributes: (1) if any words or phrases should be added, (2) if any words of phrases should be removed, and (3) what name they would chose.

At a high-level, the participants' answers to the survey questions give a rough structure to how the unique attributes need to be transformed to create a name: Addition, Removal, and Modification.

Fig. 2 shows the distribution of how the unique attributes were transformed by each participant. In the figure, the *y*-axis shows each type of transformation made by each participant (i.e., change), and the *x*-axis shows how many times each participant made each type of transformation (i.e., count). Note that

```
public void ***() throws Exception {
      RedisProcess runner = new RedisRunner()
              .nosave()
              .randomDir()
              .port(6311)
              .run();
      URL configUrl = getClass().getResource("redisson-jcache.yaml");
      Config cfg = Config.fromYAML(configUrl);
      Configuration<String, String> config = RedissonConfiguration.fromConfig(cfg);
      Cache<String, String> cache = Caching.getCachingProvider().getCacheManager()
              .createCache("test", config);
      CacheAsync<String, String> async = cache.unwrap(CacheAsync.class);
      async.putAsync("1", "2").get();
      assertThat(async.getAsync("1").get()).isEqualTo("2");
      cache.close();
      runner.stop();
}

Identified Attributes: putAsync, getAsync

Attributes Additions: Cache

Abstract of Attributes: (-put, -get) Async

Append prefix: Test

Join as Name: TestCacheAsync
```

Fig. 1. From attributes to a descriptive name.

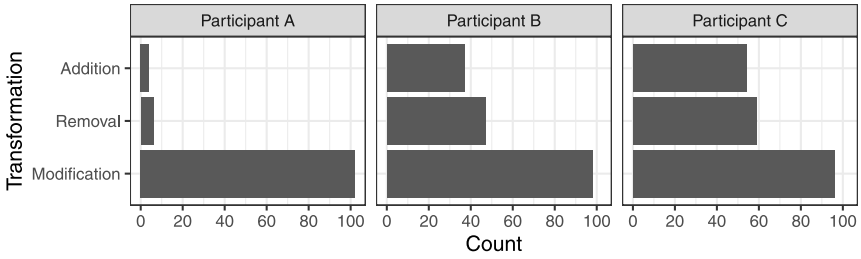

Fig. 2. Proportion of change by each participant.

multiple additions, removals, or modifications, may be done for each test. For example, Participant A made 4 additions, 6 removals, and 102 modifications while Participant B made 37 additions, 47 removals, and 98 modifications. We can make two observations based on the data shown in Fig. 2. First, for each participant, the number of additions and removals is often less than the number of modifications. This observation indicates that additions and removals are not always needed when transforming the unique attributes to a name but modifications are often needed. Second, the count of modification for each participant is often significantly greater than the number of tests—45. This observation indicates, even if there is no additions and removals made to the unique attributes, modifications are always needed to transform the unique attributes to a name.

To further understand each of the transformation type, we used an open, axial, selective coding process. Similar to the previous work (Wu and Clause, 2022a), first, each author (i.e., with at least 5 Java development experience) independently examined the additions, removals, or modifications of each considered test in the previous evaluation and tagged the portions of the test body where they believe the additions, removals, or modifications should be performed. These portions of the test body are the potential types of the additions, removals, or modifications and can be different types of code elements/statements such as method calls, constructors, and assertions. Each tag consisted of a word or short phrase that characterizes the type of addition, removal, or modification. After each author tagged each test independently, the authors together examined the tagged tests in order to reach consensus on where the additions, removals, or modifications should be performed in a test's body and to discuss the open codes. Only a few disagreements were encountered, and they are mostly related to if we should consider the mathematical expressions or some certain types of method calls (e.g., getters and setters). Moreover, after we completed the coding process, we consulted with the original participants from the previous evaluation as a sanity check. During an individual, online meeting, each participant was shown their answers and the results of the coding process. In all cases, the participants' felt that the outcome of the coding process accurately captured their intent and the rationales behind their choices. For the encountered disagreements, we also discussed them with the participants and followed the majority of the participants' opinions to resolve them. In addition, the participants expressed interest in eventually using an implementation of the name generation approach as they believe that it would help them generate acceptable names more quickly.

2.1.1. Addition

Addition occurs when a participant adds additional words or phrases to the set of attributes identified by the approach. As a result of the coding process, we identified the following types of addition:

Constructor: the name of the class under test or a constructor used in the assertions from the test body. For example, in `runWithIoExceptionDispatchRetry` from `Picasso`, `BitmapHunter` is the name of class under test and is added to the set of attributes.

Variable name: the name of a variable that contains an instance of the class under test or is used in an assertions. For example in `bitmapConfig` from `Picasso`, `data` is the name of a variable that contains an instance of the class under test that is added.

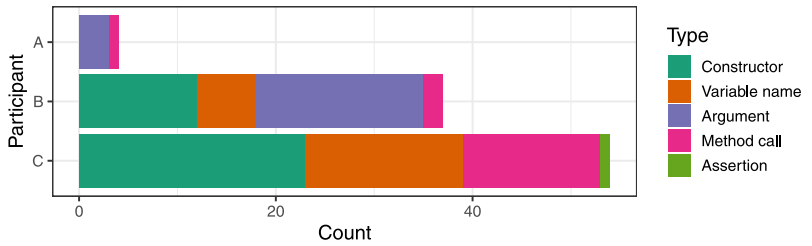

Fig. 3. Proportion of types of attribute additions. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

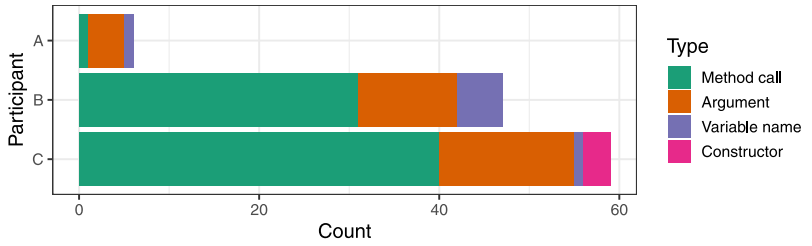

Fig. 4. Proportion of type of attribute removals.

Argument: an argument that is passed to a focal method call (Ghafari et al., 2015). For example, in `testConcurrentGetMachines` from `Sentinel`, `concurrent` is a method argument that is extracted from the focal method calls and added to the set of attributes.

Method call: a call to the method that is declared by the class under test. For example, in `nameString` from `Jedis`, `clientSetname` is a call to a method that is declared by the class under test. Often, the methods that are called most frequently are added.

Assertion: a call to an assertion used in the test body. For example, in `withoutFallbackValue` from `Moshi`, `fail` is added to the set of attributes.

Fig. 3 shows the number of times each type of addition occurred as a stacked barchart. In the figure, the *y*-axis shows each participant, the colors show the type of addition, and the *x*-axis shows how many times each participant made each type of addition. For example, from each type of addition's perspective, Participant A makes two types of attribute addition: 3 times of Parameter additions, and 1 times of Method call additions.

Based on the data from Fig. 3, it appears that what to add is a personal choice. For example, while Participant A often preferred to add Parameters and Method calls, Participant B often preferred to add Constructors and Variable names. This suggests that a fully automated name generation approach is difficult and a cooperative approach that works with developers may be preferred.

2.1.2. Removal

Removal occurs when a participant removes existing words or phrases from the set of attributes identified by the approach. As a result of the coding process, we identified the following types of removal. Note that, with the exception of Assertion, the types of removal are essentially the inverse of the types of addition.

Constructor: a constructor that is not declared by the class under test. For example, in `testInnerEntry` from `Fastjson`, `InnerEntry()` is a constructor that is not the name of the class under test that is removed from the set of attributes.

Variable name: the name of a variable that contains something other than an instance of the class under test. For, example in `testConcurrentGetMachines` from `Sentinel`, `success` is the name of a variable that contains an instance of another classes (`appInfo`) that is removed from the set of attributes.

Argument: an argument passed to a non-focal method. For example, in `test_disableCookieManagement` from `Webmagic`, `cookie` is a method argument from a non-focal method that is removed from the set of attributes.

Method call: a call to a method that is declared by a class other than the class under test. For example, in `testRemove` from `Webmagic`, `isFalse` is a call to a method declared by another class that is removed from the set of attributes.

Fig. 4 shows the number of times each type of removal occurred as a stacked barchart in the same format as Fig. 3. Like for addition, removal also appears to be a personal choice. For example, while Participant A often preferred to remove Parameters, Participant B often preferred to remove Method calls. Again, this suggests that a fully automated name generation approach is difficult and a cooperative approach that works with developers may be preferred.

2.1.3. Modification

Modification occurs when a participant modifies existing words or phrases in the set of attributes. As a result of the coding process, we identified the following types of modification:

Format: modify a necessary, but verbose attribute to add or remove pieces from it. For example, a leading or trailing "test" is often added to an attribute and syntactic elements such as keywords (e.g., `new`) or parenthesis are often removed. More specifically, we found an attribute —`new StandardCategoryURLGenerator()` generated from `JFreeChart`, a leading "test" is appended, and the `new` keyword and parentheses at the end of the constructor are removed. So the resulting attribute is: `testStandardCategoryURLGenerator`.

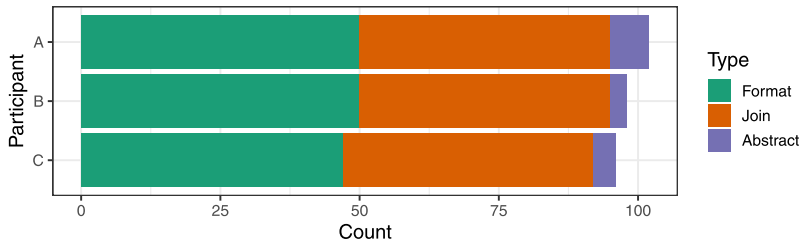

Fig. 5. Proportion of type of attribute modifications.

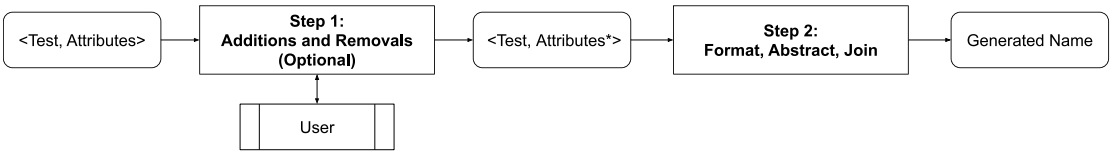

Fig. 6. Overview of approach.

Abstract: merge two or more attributes to remove duplication. For example, `getWidth` and `getHeight` would be abstracted to `WidthHeight` and `getWidth` and `setHeightWidth` would be abstracted to `Width`. More specifically, we found two attributes—`putAsync` and `getAsync` generated from `Redisson` are abstracted to `Async`; another two attributes—`providerInvoker` and `consumerInvoker` generated from `Sentinel` are abstracted to `ProviderConsumerInvoker`.

Join: the final set of attributes are joined into one as a name for the test. For example in Fig. 1, the final set of attributes—`Cache` and `Async` with a prefix `test` are joined together as the final name—`TestCacheAsync`.

Fig. 5 shows the number of times each type of modification occurred as a stacked barchart in the same format as Figs. 3 and 4. For example, Participant A made 50 Format modifications, 45 Join modifications, and 7 Abstract modifications.

Unlike for additions and removal, this data shows that modifications are much more consistent across participants. This suggests that, unlike for the other types of transformation, it is feasible to fully automate this part of the process of transforming unique attributes into unit test names.

2.2. Implementation

Based on what we learned from analyzing the transformations made by the participants, we designed the name generation approach shown in Fig. 6. As the figure shows, the input to the approach is a ⟨*Test*, *Attributes*⟩ pair where Test is the target test and Attributes are the unique attributes of the target test generated by the uniqueness-identification approach. Fig. 7 shows two tests and the output of the uniqueness-identification approach, which is used as the running example of this section. The identified attributes of the first test are: `putAsync` and `getAsync` (both are calls to methods defined by the class under test), and the identified attributes of the second test are: `providerInvoker` and `consumerInvoker` (both are variables names defined by the class under test). Those attributes are unique because no sibling (i.e., tests in the same class) contains both method calls. While the output of the approach agrees with human judgement about what features of a test make it unique, they cannot be directly used as a test name. Therefore, the output of the approach, a name for the input test, is generated in two main steps.

The purpose of the first step is to allow users of the approach to modify the set of attributes. Because we learned that additions and removals are (1) less frequent than modifications, and (2) a personal choice we made this step both optional and interactive. A user is shown the test body and the set of attributes and asked if there is anything they would like to add or remove. If they chose to make any changes, the updated set of attributes is passed to the second step.

The purpose of the second step is to modify the possibly updated attributes in order to produce a name. Unlike the first step, this step is fully automated. First, each attribute is stripped of syntactic elements like `new` or parentheses are removed. For the examples from Fig. 7, the parentheses around the method calls and variable names are trimmed for human-readability. Second, the formatted attributes are abstracted to merge similar attributes in two ways. The first way to abstract the attributes is to check whether there are common prefixes or suffixes among subsets of the attributes. To handle attributes that have a common prefix or suffix, the approach scans the attributes and grouping attributes with common prefixes and suffixes together. Then within each group, the shared text is merged into one word or phrase and the remaining text is combined without any connectors. For example, `providerInvoker` and `consumerInvoker` are abstracted to `ProviderConsumerInvoker` for the second test in Fig. 7. The second way to abstract the attributes is to check whether there are common method pairs (e.g., get/set, put/get, etc.) among subsets of the attributes. To handle the method pairs, the approach scans the attributes and grouping attributes that contain the elements of the method pairs. The complete list of method pairs is available online (Wu and Clause, 2022b). Then within each group the shared method pair is deleted and the remaining text is combined without any connectors. For example, `putAsync` and `getAsync` are abstracted to `Async` for the first test in Fig. 7. Finally, the approach joins the modified attributes by their execution order in the test body as a descriptive name. For example, the generated name for the first test is `TestCacheAsync` and for the second test is `TestProviderConsumerInvoker`. The attributes are ordered, in this case, based on execution order, and each word in the attributes are capitalized and the resulting attributes are joined together, in order, with a leading `Test`.

3. Empirical evaluation

To evaluate our approach, we conducted an empirical study of the names it generates. Because the perceived quality of a name is subjective, it is common to use qualitative methods, such as surveys, in this type of work (Wu and Clause, 2020; Moreno et al., 2013; Zhang et al., 2016; Daka et al., 2017; Sridhara et al.,

```
public void ***() throws Exception {
      RedisProcess runner = new RedisRunner()
              .nosave()
              .randomDir()
              .port(6311)
              .run();
      URL configUrl = getClass().getResource("redisson-jcache.yaml");
      Config cfg = Config.fromYAML(configUrl);
      Configuration<String, String> config = RedissonConfiguration.fromConfig(cfg);
      Cache<String, String> cache = Caching.getCachingProvider().getCacheManager()
              .createCache("test", config);
      CacheAsync<String, String> async = cache.unwrap(CacheAsync.class);
      async.putAsync("1", "2").get();
      assertThat(async.getAsync("1").get()).isEqualTo("2");
      cache.close();
      runner.stop();
}

Identified Attributes: putAsync, getAsync

Attributes Additions: Cache

Abstract of Attributes: (-put, -get) Async

Append prefix: Test

Join as Name: TestCacheAsync

public void ***() {
    ...
    providerConfig.setParameter(SentinelConstants.SOFA_RPC_SENTINEL_ENABLED, "");
    assertTrue(providerFilter.needToLoad(providerInvoker));
    RpcConfigs.putValue(SentinelConstants.SOFA_RPC_SENTINEL_ENABLED, "false");
    assertFalse(providerFilter.needToLoad(providerInvoker));
    assertFalse(consumerFilter.needToLoad(consumerInvoker));
}

Identified Attributes: providerInvoker, consumerInvoker

Abstract of Attributes: Provider (+) Consumer (-Invoker) Invoker

Append prefix: Test

Join as Name: TestProviderConsumerInvoker
```

Fig. 7. From attributes to a descriptive name.

2010). In this evaluation, we considered the names generated by our approach and several alternative approaches, including the original, developer-written test names. Each name was given to a participant who was asked to assess the name with respect to several properties such as completeness and conciseness and to give an overall judgement of its quality.

Based on the participants' responses, we answered the following research questions:

(1) RQ1: Does a participant's level of familiarity with a test impact their opinions about its name?
(2) RQ2: Are there differences between a participant's opinions of conciseness and completeness for the names generated by the approaches?
(3) RQ3: Can completeness and conciseness capture the acceptability of a test name? If not, do the participants consider other qualities?
(4) RQ4: How to understand completeness and conciseness in the context of unit test names?

We first chose to investigate whether a participant's level of familiarity with a test impacts their opinions about its name. This is important since it determines whether we can treat a participant's responses uniformly or need to keep them separate in the remainder of the evaluation. Second, we chose to look at completeness and conciseness since they are standard metrics. Completeness is important because it measures whether necessary information is included. Documentation that is incomplete may be missing critical information, which significantly reduces its effectiveness (Kipyegen and Korir, 2013; Briand, 2003). Conciseness is important because it measures whether unnecessary information is included. Documentation that contains extraneous information may waste developers' valuable time and effort of reading through them (Novick and Ward, 2006; Sridhara et al., 2010). Because test names are an important form of documentation, they should be both complete and concise. However, completeness and conciseness do not necessarily capture all of the things that are looked for in a test name. Therefore, the purpose of RQ3 is to capture other qualities that may influence the participants' opinions about the acceptability of test names. Finally, the purpose of RQ4 is to learn more about any such qualities by directly asking about the rationales the participants use when naming tests. The remainder of this section describes our participants and how we recruited them; the test names they were asked about; the alternative approaches we considered; and a discussion of the results for each research question.

3.1. Participants and subjects

As mentioned previously, our primary evaluation instrument is to survey participants. This subsection describes our selection criteria and process. As participants, we want professional developers who are experienced with a project. Choosing such developers has several benefits. First, they will be unfamiliar with our previous research or the current studies in this paper. In

```
@Test
void getBillingCurrency() {
    DailyRatedUsageLineItem lineItem = new DailyRatedUsageLineItem();
    String value = "USD";
    lineItem.setBillingCurrency(value);
    assertEquals(value, lineItem.getBillingCurrency());
}

Our Approach: TestBillingCurrency
Original Name: getBillingCurrency
Code2Vec: pay
Host Approach: lineItemString
```

Fig. 8. Comparison between the generated name from our approach and alternatives.

Table 2
Considered projects for evaluation.

Project	Version	LoC	# Tests	LoH	# Contr.
Guava	368c337	400,801	13,962	11	280
Guice	9b371d3	183,049	1,280	9	64
Octane-gocd-plugin	f3cc22b	6,190	26	5	8
Partner Center SDK	cb7c53f	51,490	261	2	29
Robodriver	8286b8e	5,572	82	3	14

that case, we can eliminate the possibility of favoritism towards our approach and collect unbiased responses from them. Second, prior work has shown that students are not an analogue for professional developers (Beller et al., 2015). This matters to us for two main reasons. For one, while prior work has used students to evaluate aspects of naming tests (e.g., Zhang et al., 2016; Daka et al., 2017; Wu and Clause, 2022a), no one has yet considered experienced developers. As domain knowledge and level of familiarity with a project may influence perception of names, the validity and generalizability of the evaluation may be impacted if we were to use students.

In order to recruit participants that meet our criteria, we started with finding people rather than projects. Experience with prior evaluations has shown that "cold calling" or unsolicited requests have a very low response rate. Rather than attempt this again, we chose to leverage our professional network (i.e., previous work experience and connections at top-tier technology companies) to meet with team leads or associate managers. If they agreed to work with us, we asked them to nominate a senior, staff, or principle developer from a project they oversee. If their nominee agreed, they, along with their project, became a participant. As a result of this process, we recruited a group of 5 professional software developers (i.e., senior to principle level), one from each of the projects shown in Table 2. In Table 2, LoC indicates the lines of code, LoH indicates the length of history, and # Contr. indicates the estimated number of contributors. The professional software developers are the experts of their different focuses of software development and come from 3 top-tier technology companies. Some of them launched secure, back-end software products for millions of global users, and others developed real-time, web-based applications for both experienced cloud developers and server administrators.

3.2. Considered approaches

To compare the generated names from our approach with the alternatives, we implemented a proof-of-concept prototype of our uniqueness-based, name generation approach to be used in the empirical evaluation. This prototype of our approach is also available online (Jianwei Wu, 2022a). When using the prototype of our approach, the names that were generated for this evaluation did not have any additions or removals in their name generation process (i.e., this part was disabled). We decided to implement this for removing a source of bias and the consistency in what the participants assessed. It is considered as a worst-case scenario from the prototype's perspective since an important part of functionality is disabled. We also considered three alternative name generation approaches: the original, developer-written names, Høst's approach (Høst and Østvold, 2009), and Code2vec (Alon et al., 2019). We chose the original names because they were manually written by developers experienced with the project. As such, they are the closest we have to "correct" names. The other approaches were selected because they represent two types of name generation strategies. We chose Høst's approach because it is well-studied, rule-based approach. Because there is no existing implementation of Høst's approach, we created our own. To do this, we used the relevant naming rules from Høst's publication, which describe naming bugs and name-specific rules. We used their examples of poor names and the corresponding replacement names to verify the validity of our implementation. This implementation of Høst's approach is available online (Jianwei Wu, 2022b). We chose Code2vec because it is a more modern, machine learning-based approach (Alon et al., 2019). The implementation of Code2vec we used is a Commit-c98e8f7 (Uri Alon and Yahav, 2022).

For an example of comparing the generated name from our approach with the alternative approaches, the test in Fig. 8 is originally named as `getBillingCurrency`. Code2Vec provides a name for this test as `pay` that only makes a local sense for this specific test method, and Høst's approach provides a better name than Code2Vec but still fails to capture the testing goal of this test. Because the test body of this test includes both `setBillingCurrency` and `setBillingCurrency`, our approach generated a descriptive name as `TestBillingCurrency` that is more comprehensive than the original name.

3.3. Survey questions

Each participant was given a survey based on their test set. For each test, the participant is first shown the test body with a brief description of the question. Second, the participant is asked to indicate their level of agreement with the statements: "⟨*name*⟩ is complete", "⟨*name*⟩ is concise", and "⟨*name*⟩ is acceptable" where ⟨*name*⟩ is the name generated by each approach for the test body. Their responses are collected on a Likert scale from 1 to 5, where 1 indicates Strong Disagreement and 5 indicates Strong Agreement. For example in Fig. 9, the Likert scale is set to measure the level of agreement for each generated name for the shown test body from our approach and alternatives. And the order of the generated names is randomized with true randomness to avoid potential bias from the participant. Finally, we asked the participant to write a few sentences to further explain their responses (i.e., as the last question for each test in a text box). In particular, we asked them to detail the naming rationales

Q1.A

Please indicate your level of agreement with the following statements. Here, complete means that the candidate includes everything that the name of the following test should include:

```
public void XXX() {
  assertEqualsBothWays(mapStringInteger, Types.mapOf(String.class, Integer.class));
  assertEqualsBothWays(listString, Types.listOf(String.class));
  assertEqualsBothWays(setString, Types.setOf(String.class));
}
```

	Strongly disagree				Strongly agree
`testListSetMap` is complete	○	○	○	○	○
`equalsBothWays` is complete	○	○	○	○	○
`testSetListMap` is complete	○	○	○	○	○
`testEquality` is complete	○	○	○	○	○

Fig. 9. Example of survey questions.

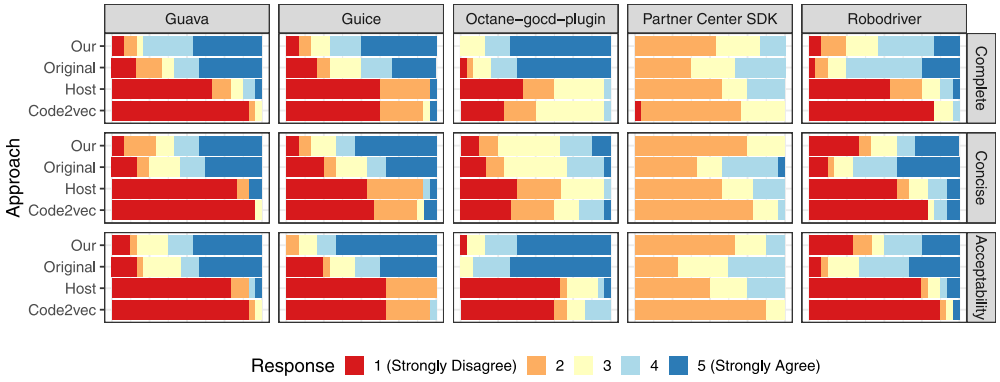

Fig. 10. Participant agreement with completeness, conciseness, and acceptability statements. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

behind their choices. There was no time limit for the survey, so participants were able to take as much time as they wanted to answer each question. A complete set of the survey questions and collected survey data can be found in Wu and Clause (2023).

3.4. Data collection

Because evaluating a test name is relatively time consuming, it is infeasible to assess all 15,611 tests from the considered projects. Instead, we created a custom set of 24 tests for each participant. A pilot study with the authors indicated that evaluating a test would take a little over a minute, and 30 min is a reasonable time commitment. Each participant's test set contains 12 tests randomly chosen from their project and 3 tests randomly chosen from each other project (12 total). Including both tests from their project and from other projects allows us to assess whether their opinions differ based on their level of familiarity.

Fig. 10 shows the collected responses for the completeness, conciseness, and acceptability questions in a series of 15 barcharts, one for each combination of participant/project (columns) and statement type (rows). In each barchart, the *y*-axis shows each considered approach, the *x*-axis shows the percentage of responses in each Likert category, and the color of each bar indicates the response on a diverging scale from red (Strong Disagreement) to blue (Strong Agreement). For example, the topmost, left-most barchart shows the responses from the participant who developed `Guava` when asked about the completeness of the test names. As the chart shows, the participant's response when asked about completeness was Strongly Agree ≈46% of the time for our approach, ≈42% of the time the original names, ≈4% of the time for Høst's approach, and ≈0% of the time for Code2vec. Similarly, the same participant's response when asked about completeness was Strongly Disagree ≈8% of the time for our approach, ≈16% of the time the original names, ≈66% of the time for Høst's approach, and ≈91% of the time for Code2vec.

3.5. RQ1: Does a participant's level of familiarity with a test impact their opinions of its name?

The purpose of our first research question is to investigate whether a participant's level of familiarity with a test impacts their opinions about its name. The outcome of this question determines if we can treat all a participant's responses uniformly in the rest of this evaluation; if so, we no longer need to separate this evaluation by level of familiarity. To check for a difference in the responses, we performed a series of Mann–Whitney Wilcoxon tests (Dunn, 1964; Pappas and DePuy, 2004). We chose to use the Mann–Whitney Wilcoxon test because we have one nominal

variable (own project or not), one measurement value (the participant's responses), and we do not know whether our data are normally distributed. We chose an alpha (α) of 0.05, and used R version 4.1.1's implementation of the test (i.e., wilcox.test). The full test results are available online (Jianwei Wu, 2022c).

In total, we ran 15 tests results: 3 types of statement (i.e., complete, concise, and acceptable) $\times$ 5 projects. The resulting p-values range from $\approx$0.11 to $\approx$0.94. For example, the p-values for the tests comparing the participant from Guava's responses is $\approx$0.49 for complete, $\approx$0.87 for the concise statements, and $\approx$0.74 for acceptable. Because all of the p-values are above 0.05, we conclude that there is not a statistically significant difference between the responses for each participant's own project and for other projects. This means that it is unlikely that a participant's level of familiarity with a test impacts their opinions of it completeness, conciseness, and acceptability. Therefore, we treat all of the responses uniformly in the rest of this evaluation rather than separate them by familiarity.

3.6. RQ2: Are there differences between the participant's opinions of conciseness and completeness for the names generated by the approaches?

The purpose of our second research question is to investigate whether there are differences between the participant's opinions of conciseness and completeness for the names generated by the considered approaches. To answer this research question, we use the data presented in the charts in the top two rows in Fig. 10. Based on a visual observation of these charts, we can make several observations:

First, it appears that the overall distribution of responses is similar for conciseness and completeness across all participants/projects. Second, the participants' opinions about the original names and the names generated by our approach are similar and generally positive (i.e., the majority of the responses are on the Strongly Agree side of the scale) while the opinions for the names generated by Høst's approach and Code2vec are also similar but generally negative (i.e., the majority of the responses are on the Strongly Disagree side of the scale). The only outlier to these trends are the responses from the participant from `Partner Center SDK` shown in the fourth column. Overall, this participant's responses are more neutral (i.e., close to the middle of the scale) and there is not a distinct bifurcation between the responses for our approach and the original name and the responses for Høst's approach and Code2vec. We believe that these differences are due to several factors. First, `Partner Center SDK` is a Java SDK, rather than a utility library or plugin, which might necessitate a different set of test expectations. Second, this project is developed to support different programming languages and technologies (i.e., ASP.NET, REST), so the developers may bring some testing conventions used for other languages. Because all of the name generation approaches are geared towards Java unit tests, they might not meet all of the expectations from the other domains.

To further investigate our visual observations, we again performed a series of Wilcoxon rank sum tests. In this case though, we used pairwise tests with Bonferroni correction. Pairwise tests are necessary as the nominal value (the considered approaches) has more than two categories and Bonferroni correction adjusts for performing multiple tests. We again chose an alpha (α) of 0.05, and used R version 4.1.1's implementation of the test (i.e., pairwise.wilcox.test). The full test results are available online (Jianwei Wu, 2022c).

In total, we ran 10 tests: 2 statement types (complete and concise) $\times$ 5 projects. Each test produced 6 p-values, one for each distinct combination of considered approaches (i.e., $\binom{4}{2}$), for a total of 60 p-values. The resulting p-values range from $\approx 1.1 \times 10^{-8}$ to $\approx$1 with 28 greater than 0.05 and 32 below 0.05. These results confirm our visual observations. For all participants, the names generated by our approach are viewed as comparable (i.e., equivalent) to the original test names in terms of completeness and conciseness. Conversely, the names generated by Høst's approach and Code2vec, while comparable to each other, are viewed as significantly less concise and complete by the participants. Overall, this a positive result for our approach. It is capable of generating test names that are as complete and concise as the original names.

3.7. RQ3: Can completeness and conciseness capture the acceptability of a test name? if not, do the participants consider other qualities?

The purpose of RQ3 is to investigate the participants' opinions about the overall acceptability of the test names. While completeness and conciseness are important qualities, it is possible that the do not capture all of the qualities that are important when naming tests. By asking about acceptability directly, we can determine if there are important aspects of a test name that are not captured by completeness and conciseness.

To answer this research question, we use the data presented in the charts in the bottom row in Fig. 10. Based on a visual observation of these charts, the responses for acceptability appear to be similar to the responses for completeness and conciseness. To investigate this observation further, we used a procedure similar to the one used for RQ2: a series of pairwise Wilcoxon tests with Bonferroni correction ($\alpha = 0.05$, full test results are available online Jianwei Wu, 2022c). In this case, the resulting 30 p-values (1 statement types (acceptability) $\times$ 5 projects $\times$ 6 distinct approach combinations) range from $\approx 2.3 \times 10^{-8}$ to $\approx$1 with 12 p-values above the 0.05, and 18 below the 0.05. Again, results of the statistical tests confirm our visual observations. Our approach is similar to the original names; Høst's approach is similar to Code2vec; and both our approach and the original names are distinct from Høst's approach and Code2vec. This result is also positive for our approach: the names it generates are similar to the original names in terms of acceptability and are significantly better than the names generated by the alternative approaches. Moreover, the data suggests that, because the overall distributions are similar, there are no qualities beyond completeness and conciseness that a serious concern when naming tests.

3.8. RQ4: How to understand completeness and conciseness in the context of unit test names?

The purpose of our last research question is to better understand completeness and conciseness in the context of unit test names. While the RQ3 suggests that completeness and conciseness are the primary properties that a test name generation approach should consider, it is not clear what concrete aspects of a test map to completeness and conciseness. To investigate this research question, we used the participants' free response answers where we asked them specifically about the rationales behind their choices. We used a combination of axial and selective coding to analyze the responses along two categories: *completeness* and *conciseness*.

As a result of this process, we concluded several things about the participants' views on completeness and conciseness with respect to test names: A *complete* test name should include:

the function name that is being tested and what aspect of the function is tested. Two of the participants mentioned

this rationale. Their judgement of completeness is heavily influenced by if a name contains the tested function name and what aspect of the function is tested (e.g., focal method and its testing scenario).

the class name (i.e., class under test). Two of the participants mentioned this rationale. Their judgement of completeness is heavily influenced by if a name contains the name of the class under test. They stated that having the name of the class under test can help people locate the class if there is a bug found in the process of unit testing.

the assertions in the test body. Two of the participants mentioned this rationale. Their judgement of completeness is heavily influenced by if the presented name contains the test's assertions. They explained their reasoning behind this rationale is to use the final state and context in the assertions to provide descriptive test names.

the actual behavior of the test case. One of the participants mentioned this rationale. Their judgement of completeness is heavily influenced by if the presented name contains the actual behavior of the test case. The actual behaviors of the test could be various, such as calling a specific method to test a software feature (e.g., method calls), testing a certain parameter in a series of assertions (e.g., assertion's parameters), or initiating an object from the class under test (e.g., variables).

and a *concise* name should:

be descriptive, not generalized. Three of the participants mentioned this rationale. Their judgement of a test name being concise is heavily influenced by if the presented name is descriptive.

summarize the test and exclude irrelevant information. One participant mentioned this rationale. Their judgement of a test name being concise is heavily influenced by if the presented name can summarize the context of the test and exclude irrelevant information.

Overall, the participants' have a consistent view of what concrete aspects of a name contribute to it being complete and concise. These views also help explain why the names generated by our approach score highly while the names generated by Code2vec and Høst's approach perform poorly. First, the names generated by our approach incorporate with the views of completeness. For *the function name that is being tested and what aspect of the function is tested*, our name generation approach works well with respect to this view as it is designed to extract the focal method (function name that is being tested) and its parameters (what aspect of the function) from the test body. For *the class under test name*, our name generation approach works well with respect to this view as it is designed to extract the name of the object under test (class name from the class under test) from the test body. For *the assertions in test body*, our name generation approach works well with respect to this view as it is designed to extract both the assertion calls and their parameters from the test body. For *the actual behavior of test*, our name generation approach works well with respect to this view as it iterates through all unique attributes (method calls, assertions, and variables) from the test body to capture the behavior of the test. Second, the names generated by our approach also incorporate with the views of conciseness. For *be descriptive*, the goal of our name generation approach directly incorporates with this high-level view, which is to provide descriptive test names. For *summarize the test and exclude irrelevant information*, the extracted attributes of a test sometimes do contain redundant information, so this view is not always met. However, it is not an issue for our name generation approach. Our approach have a Removal step to exclude the unnecessary attributes when generating names, and its prototype implementation allows users to manually inspect the existing attributes and remove any of them if deemed necessary (Jianwei Wu, 2022a). Additionally, our approach summarize every attribute of the test when generating descriptive name as part of the Abstract step. Overall, the generated names from our approach can support this view and produce concise test names that meet developers' needs.

Existing machine learning and model-based approaches like Code2vec can sometimes capture the uniqueness of test, but are not focused on that. Therefore, their approaches are less successful in generating test names that can meet developers' needs (i.e., a name should be descriptive — complete and concise). Existing rule-based approaches like Høst's approach can occasionally produce descriptive names, but are highly relied on the robustness of their naming rules. If their naming rules fail, there could be important information missing in their names or irrelevant information added to their names. Moreover, because their naming rules are often strict and programmatic, there is no obvious way to exclude irrelevant information without violating one or more rules. Therefore, the generated names from their approaches are less likely to be complete and concise at the same time.

3.9. Demographics, insights, and practicality

As an attempt to correlate the demographics of the participants' expertise with the results of completeness, conciseness, and acceptability, we also briefly investigated such correlation. Because the team leads or associate managers who agreed to join our study randomly selected each participant with a project, not us, we cannot directly refer to each participant's past development experience. Therefore, we used the project attached to each participant as a potential source of reference to their background. However, similar to what we can observe in Fig. 10, there is no significant difference in the participants' judgement between something out of their expertise and in their expertise. We believe that professional developers often have experience in different areas and phases of software development and are consistent when judging unit test names out of or in their expertise. The collected metadata can be found online (Wu and Clause, 2023).

Based on the participants' opinions and Mann–Whitney Wilcoxon test results from the research questions in Section 3, we summarized the following core insights.

professional developer's level of familiarity with a test is unlikely to impact their opinions of it completeness, conciseness, and acceptability of unit test names. For future research effort, a larger group of professional developers can be introduced and invited to evaluation without any concern with data validity, and all collected data from the survey can be treated uniformly.

conciseness and completeness can often capture the acceptability of a JUnit test name. For future software development effort, conciseness and completeness can be used to develop automated test or name generation tooling with confidence to meet developers' needs. For future research effort, conciseness and completeness can be tested and extended to other object-oriented programming languages (e.g., C++, Golang, Kotlin, etc.), so a unified test naming framework might eventually be established across different languages.

Furthermore, based on the summarized insights, we provide several real-world scenarios that show how our approach fit into developers' daily workflow.

Integrate our approach with continuous integration and continuous delivery (CI/CD). Our approach can be part of the continuous integration and continuous delivery process (CI/CD) and provide guidance for developers in their daily workflow. For example, the generated names from our approach can be automatically presented to developers in any CI/CD workflow such as TeamCity or GitHub Actions. When a new commit or pull request is pushed or merged into main branch, developers should compare the proposed test names with the generated ones and decide if they should improve them.

Integrate our approach with development tools. Our approach can also be implemented as an integrated development environment (IDE) plugin for IntelliJ Idea, Visual Studio Code, or Eclipse. For example, when developers try to deal with a project with legacy Java code (i.e., the original developers of it left the team a long time ago), they can launch our tool conveniently from their IDEs and use it to generate descriptive names for the existing JUnit tests. Understanding from the generated name of each test, developers can decide if they would prefer to replace the test name, re-write the test body, or remove the test case entirely. Therefore, the generated names from our approach serve as a summary of the existing tests and can significantly reduce the amount of time and effort that developers need to spend on refactoring the legacy JUnit tests.

Integrate our approach with code reviews. When performing code reviews, developers can utilize the implementation of our approach as a command-line tool to improve existing JUnit test names. For example, when a developer (i.e., contributor) submits a pull request to their Git-like version control system (e.g., GitHub, Mercurial, Subversion, etc.) and requests reviews from other developers, the other developers (i.e., reviewers) can run our approach against the submitted JUnit tests. If the reviewers notice and deem any test names that are not descriptive and should be replaced, they can require change from the contributor by proposing the generated names from our approach.

Implement our approach for other programming languages. Because our approach are flexible as mentioned the previously mentioned scenarios and can meet developers' approval, it can also be extended to other programming languages. For example, from our preliminary discussion with some developers after the evaluation in Section 3, they mentioned that their manually written Golang and Kotlin unit tests generally follow the same naming pattern (i.e., uniqueness). And some of their teams considered the uniqueness-based naming pattern as the preferred practice and formalized it in their software development handbook for new hires to follow.

4. Threat to validity

Based on the results of the empirical evaluation, our approach perform well in the empirical evaluation and can generate test names that are as complete and concise as the original names. However, this does not indicate that our approach is perfect for any occasions. In this section, we looked into the four types of threats to validity.

First, for internal threat, we provided a set of 24 tests for each participant to evaluate the performance of our approach and alternative approaches. Comparing to the alternative approaches, the generated names from our approach are close to the original names and often follow participants' commonly used naming rationales. However, the considered tests are limited to a certain amount, so they might not be able represent all types of tests. To mitigate this threat, we used a random number with true randomness to select each considered test; if a test is selected twice, a different random test would be selected. Therefore, even with a limited set of tests, we believe that our considered tests can represent many different types of tests, and the generated names from our approach can generally meet developers' needs.

Second, for external threat, we tried our best to eliminate all possible bias that might be introduced to the approach or empirical evaluation. However, since the participants in our empirical evaluation is limited to 5, and the criterion of each participant might not be consistent. To mitigate this threat, none of the participants knew our research or this study before they completed the survey to ensure their judgement is unbiased and valid. For each question and participant in the each part of the survey, the order of the presented names is scrambled randomly. Moreover, rather than we chose a main contributor from the related team, we asked the invited team leads or associate managers to choose one for us, which further reduced the likelihood of favoritism from both our participants and us. Therefore, we believe that the participants can provide valid judgement towards evaluating the generated names from both our approach and alternatives.

Third, for construct validity, in order to analyze the statistical result of the empirical evaluation, we performed a series of Mann–Whitney Wilcoxon tests. However, because there are several comparable non-parametric tests (e.g., the Kolmogorov–Smirnov test Fasano and Franceschini, 1987, the Student's t-test Livingston, 2004, etc.), some better non-parametric tests could exist. To mitigate this threat, we performed a literature research to make sure the Mann–Whitney Wilcoxon tests are the best fit. To our best knowledge, existing work does show the Mann–Whitney Wilcoxon tests are fundamentally better when measuring the difference between data in human-involved studies (Kühnast and Neuhäuser, 2008). Therefore, we believe that the Mann–Whitney Wilcoxon tests fit our study and can accurately analyze the statistical result of comparing the generated names from our approach and alternatives.

Last, for conclusion validity, we provided some insights based on the statistical result of the empirical evaluation. Although our survey questions can quantitatively measure the performance of our approach and alternatives, they might not be in the preferred format by the participants. Some participants might prefer to express their own opinions when comparing the generated names from our approach and alternatives. To mitigate this threat, we added an additional question that asked the participants about how to understand completeness and conciseness in the context of unit test names, and it was discussed as RQ4 in Section 3. Therefore, we believe that the conclusion of our study is justifiable and can capture the actual thoughts from the participants about how to compare the generated names from our approach and alternatives and understand completeness and conciseness in the context of unit test names.

5. Related work

In this paper, we propose a uniqueness-based name generation approach to provide descriptive test names that meet developers' approval. The approach utilized a combination of static program analysis and formal concept analysis to identify which combination of attributes is unique to the test under consideration. Based on the identified attributes, the approach further provides an optional, interactive step of attribute additions and removals, then it performs a series of automated modifications to transform the updated attributes into a descriptive name. In this section, we discuss three areas of related work. First, previous works tried to solve the naming problem of unit tests by providing suggestions to improve existing tests. Second, other previous

works managed to propose machine learning-based, coverage goal-based, or search-based approaches to automatically generate unit tests. Finally, we also discuss several recent studies on the problem of unit test or method naming.

5.1. Techniques to suggest better test/method names

As one of the pioneers on the naming problem of tests/methods, Høst and Østvold proposed a rule-based approach to find erroneous names and suggest more suitable names for general Java methods (Høst and Østvold, 2009). Another work is to suggest accurate method names by a probabilistic language model (Allamanis et al., 2015). Overall, their approach performed well when debugging or suggesting Java method names. However, because their rule-based or probabilistic language model are limited to work on several specific types of tests or methods, their provided suggestions might not always meet developers' approval. This assumption is confirmed with the empirical evaluation in Section 3. Other existing studies tried to access the naming problem in methods/tests by evaluating the quality of them (Zhang et al., 2014; Alsuhaibani et al., 2022; Moreno et al., 2013; He et al., 2011; Zhou et al., 2019; Butler et al., 2010; Olney et al., 2016; Pollock et al., 2007; Butler et al., 2011). All of these approaches managed to either provide appropriate guidance to improve non-descriptive method names or generate accurate summaries for Java methods or classes. However, one major limitation of their work is that developers often have a different mindset when naming their unit test from naming general Java methods or classes. The limitations mentioned above prevent their approaches from being useful for suggesting descriptive test names without some heavy modifications.

5.2. Techniques to generate test/test names

To accurately generate test/method names, Alon et al. proposed a machine learning-based technique to automatically generate accurate names for general Java methods (Alon et al., 2018, 2019). Another well-known type of technique is to generate test name using a set of API-level coverage goals (Daka et al., 2017). Similarly, Zhang et al. used a combination of natural-language program analysis and text generation to generate descriptive test names (Zhang et al., 2016). These techniques are good in term of providing descriptive names under specific circumstances (e.g., fit in their training set, a matched coverage goal, a perfect generation process, and etc.) and should be applicable to unit tests. Nonetheless, for the countless other types of projects outside their specific circumstances, their generated names tend to substantially lose both descriptiveness and developers' approval.

Moreover, existing work also tried to solve the naming problem by providing fully automated approaches to generate unit tests (Fraser and Arcuri, 2011; Thummalapenta et al., 2009; Arcuri and Fraser, 2016; Taneja and Xie, 2008; Arcuri et al., 2014; Anand et al., 2013; Jensen et al., 2013; Kamimura and Murphy, 2013). Although the name generation approaches can automatically provide better and descriptive names, there are two limitations for them to become more successful at providing descriptive test names with developers' needs. First, these techniques combined the detection of what should be in a good name with the name generation process, and it significantly reduced their approaches' flexibility. Developers often preferred to add or remove elements when using a name generation approach to construct a test name. This preference is observed in the empirical evaluation from Section 3. Second, these techniques performed all name generation steps (i.e., generating both test names and bodies) in one click and used their own design and definitions to produce the entire unit test. Unfortunately, when developing a project, developers (i.e., especially for senior to principle developers) often have a pre-defined set of business goals and a unit/integration test plan for their projects. There is often a mismatch between these techniques' goals of generating a test and developers' business goals and test plan. The mismatch is observed in the empirical evaluation from Section 3. More importantly, changing the business goals and test plan in the middle of development circle is often not preferred, sometimes forbidden.

Because of the limitations and mismatch between these techniques and developers, these name generation techniques cannot be applied to the vast majority of existing tests that are often manually written and maintained by developers and are significantly less effective to generate descriptive names if the mismatch exists. Therefore, we propose a developer-oriented, flexible name generation approach to provide test names that meet developers' needs.

5.3. Recent techniques on test/method naming

A more recent work used grammar patterns to understand test names and utilized their findings to recommend better names (Peruma et al., 2021). Gao et al. proposed a functional description-based approach to generate better method names (Gao et al., 2019). Kasegn and Abebe proposed a spatial locality-based approach that can recommend method names by using identifier names as a concept (Kasegn and Abebe, 2021). Nguyen et al. proposed a machine learning approach to check the consistency between the name of a given method and its implementation (Kasegn and Abebe, 2021). These approaches might work well on each specific field. However, some of them did not performed an empirical study or evaluation with developers, and others' evaluation only included arbitrary developers' responses from open-source projects on Github. Because Github is an open-source platform and public to everyone, there is no guarantee that their collected responses would still be valid if reviewed by professional/core developers from the related projects. An empirical evaluation is needed to verify these approaches with professional developers.

Yamanaka et al. proposed a new technique to accurately recommend extract method refactoring by a set of predicted method names (Yamanaka et al., 2021). However, their predicted method names were generated by Code2seq, a older version of Code2vec (Alon et al., 2018, 2019). Based on the performance of Code2vec in Section 3, we believe that this approach might not be effective for providing descriptive test names. In a recent empirical study of test generation approaches (Evosuite Fraser and Arcuri, 2011, etc.), developers have no preference towards the generated tests, and there is always some code refactoring needed for them and their names (Herculano et al., 2022). Based on their findings, a cooperative approach is preferred by developers.

6. Conclusion

In this paper, we propose a uniqueness-based name generation approach that can provide developer-approved, descriptive names for JUnit tests. Our approach is based on an existing technique (previously built by us) that can identify uniqueness of JUnit tests (Wu and Clause, 2020) and extend its functionality to generate descriptive names. In order to evaluate if the generated names from our approach meet developers' needs, we invited a group of 5 professional developers from top-tier technology companies as participants. We asked the participants to evaluate the generated names from our approach and alternative approaches in an empirical survey. Based on the participants' responses, the generated names from our approach outperformed other alternatives in terms of the generated name being complete, concise,

and acceptable, which are generally at the same level as the original names. This indicates that the generated names from our approach meet developers' approval and are more likely to be useful in practice. For our planned future work, we are going to further extend the uniqueness-based name generation approach to a self-learning, adaptive name generation approach, and perform a large-scale, empirical survey with at least 20 professional developers to evaluate the adaptive approach.

CRediT authorship contribution statement

Jianwei Wu: Conceptualization, Methodology, Data curation, Software, Visualization, Writing – original draft, Writing – review & editing, Formal analysis, Empirical study. **James Clause:** Conceptualization, Software, Writing – review & editing, Formal analysis, Supervision, Empirical study.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The research data of this paper is provided in the links of this manuscript.

References

Allamanis, M., Barr, E.T., Bird, C., Sutton, C., 2015. Suggesting accurate method and class names. In: Proceedings of the Joint Meeting on Foundations of Software Engineering. ACM, Lombardy, Italy, pp. 38–49.

Alon, U., Levy, O., Yahav, E., 2018. Code2seq: Generating sequences from structured representations of code. CoRR 1–22, arXiv:1808.01400, URL http://arxiv.org/abs/1808.01400.

Alon, U., Zilberstein, M., Levy, O., Yahav, E., 2019. Code2vec: Learning distributed representations of code. Proc. ACM Programm. Lang. 3 (POPL), 40.

Alsuhaibani, R.S., Newman, C.D., Decker, M.J., Collard, M.L., Maletic, J.I., 2022. An approach to automatically assess method names. arXiv preprint.

Anand, S., Burke, E.K., Chen, T.Y., Clark, J., Cohen, M.B., Grieskamp, W., Harman, M., Harrold, M.J., Mcminn, P., Bertolino, A., et al., 2013. An orchestrated survey of methodologies for automated software test case generation. J. Syst. Softw. 86 (8), 1978–2001.

Arcuri, A., Fraser, G., 2016. Java enterprise edition support in search-based junit test generation. In: Proceedings of the International Symposium on Search Based Software Engineering. Springer, Raleigh, North Carolina, USA, pp. 3–17.

Arcuri, A., Fraser, G., Galeotti, J.P., 2014. Automated unit test generation for classes with environment dependencies. In: Proceedings of the ACM/IEEE International Conference on Automated Software Engineering. ACM, pp. 79–90.

Beller, M., Gousios, G., Zaidman, A., 2015. How (much) do developers test? In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. 2, IEEE, pp. 559–562.

Briand, L.C., 2003. Software documentation: how much is enough? In: Seventh European Conference OnSoftware Maintenance and Reengineering, 2003. Proceedings.. IEEE, pp. 13–15.

Butler, S., Wermelinger, M., Yu, Y., Sharp, H., 2010. Exploring the influence of identifier names on code quality: An empirical study. In: Proceedings of the European Conference on Software Maintenance and Reengineering. IEEE, Madrid, Spain, pp. 156–165.

Butler, S., Wermelinger, M., Yu, Y., Sharp, H., 2011. Improving the tokenisation of identifier names. In: Proceedings of the European Conference on Object-Oriented Programming. Springer, pp. 130–154.

Daka, E., Rojas, J.M., Fraser, G., 2017. Generating unit tests with descriptive names or: Would you name your children thing1 and thing2? In: Proceedings of the SIGSOFT International Symposium on Software Testing and Analysis. ACM, Santa Barbara, CA, USA, pp. 57–67.

Dunn, O.J., 1964. Multiple comparisons using rank sums. Technometrics 6 (3), 241–252.

Fasano, G., Franceschini, A., 1987. A multidimensional version of the Kolmogorov–Smirnov test. Mon. Not. R. Astron. Soc. 225 (1), 155–170.

Fraser, G., Arcuri, A., 2011. EvoSuite: Automatic test suite generation for object-oriented software. In: Proceedings of the SIGSOFT Symposium and the European Conference on Foundations of Software Engineering. ACM, Szeged, Hungary, pp. 416–419.

Gao, S., Chen, C., Xing, Z., Ma, Y., Song, W., Lin, S.-W., 2019. A neural model for method name generation from functional description. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering. SANER, IEEE, pp. 414–421.

Ghafari, M., Ghezzi, C., Rubinov, K., 2015. Automatically identifying focal methods under test in unit test cases. In: Proceedings of the International Working Conference on Source Code Analysis and Manipulation. IEEE, Bremen, Germany, pp. 61–70.

He, N., Rümmer, P., Kroening, D., 2011. Test-case generation for embedded simulink via formal concept analysis. In: Proceedings of the Design Automation Conference. IEEE, pp. 224–229.

Herculano, W.B., Alves, E.L., Mongiovi, M., 2022. Generated tests in the context of maintenance tasks: A series of empirical studies. IEEE Access 10, 121418–121443.

Høst, E.W., Østvold, B.M., 2009. Debugging method names. In: Proceedings of the European Conference on Object-Oriented Programming. Springer, Genoa, Italy, pp. 294–317.

Jensen, C.S., Prasad, M.R., Møller, A., 2013. Automated testing with targeted event sequence generation. In: Proceedings of the International Symposium on Software Testing and Analysis. ACM, pp. 67–77.

Jianwei Wu, J.C., 2022a. Implementation of Uniqueness-based Name Generation Approach. Accessed: 2022-09-12, https://github.com/Jianwei-Wu-1/NameGenerationPlugin.

Jianwei Wu, J.C., 2022b. Lite implementation of Høst's approach. Accessed: 2022-09-12, https://github.com/Jianwei-Wu-1/HostsApproachLite.

Jianwei Wu, J.C., 2022c. Mann-Whitney Wilcoxon test Data. Accessed: 2022-10-10, https://docs.google.com/spreadsheets/d/1DE7TMNG5fX0p_-Oh33yS0rzbnt6UW8aD_SEd5bj1jPo/edit?usp=sharing.

Kamimura, M., Murphy, G.C., 2013. Towards generating human-oriented summaries of unit test cases. In: 2013 21st International Conference on Program Comprehension. ICPC, IEEE, pp. 215–218.

Kasegn, S.A., Abebe, S.L., 2021. Spatial locality based identifier name recommendation. In: 2021 International Conference on Information and Communication Technology for Development for Africa (ICT4DA). IEEE, pp. 65–70.

Kipyegen, N.J., Korir, W.P., 2013. Importance of software documentation. Int. J. Comput. Sci. Issues (IJCSI) 10 (5), 223.

Kühnast, C., Neuhäuser, M., 2008. A note on the use of the non-parametric wilcoxon-mann-whitney test in the analysis of medical studies. GMS German Med. Sci. 6.

Lison, P., 2015. An introduction to machine learning. Lang. Technol. Group (LTG) 1 (35), 291.

Livingston, E.H., 2004. Who was student and why do we care so much about his t-test? 1. J. Surg. Res. 118 (1), 58–65.

Moreno, L., Aponte, J., Sridhara, G., Marcus, A., Pollock, L., Vijay-Shanker, K., 2013. Automatic generation of natural language summaries for java classes. In: 2013 21st International Conference on Program Comprehension. ICPC, IEEE, pp. 23–32.

Nguyen, S., Phan, H., Le, T., Nguyen, T.N., 2020. Suggesting natural method names to check name consistencies. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. pp. 1372–1384.

Novick, D.G., Ward, K., 2006. What users say they want in documentation. In: Proceedings of the 24th Annual ACM International Conference on Design of Communication. pp. 84–91.

Olney, W., Hill, E., Thurber, C., Lemma, B., 2016. Part of speech tagging java method names. In: Proceedings of the International Conference on Software Maintenance and Evolution. IEEE, pp. 483–487.

Pappas, P.A., DePuy, V., 2004. An overview of non-parametric tests in SAS: when, why, and how. Paper TU04. Duke Clin. Res. Inst. Durham 1–5.

Peruma, A., Hu, E., Chen, J., AlOmar, E.A., Mkaouer, M.W., Newman, C.D., 2021. Using grammar patterns to interpret test method name evolution. In: 2021 IEEE/ACM 29th International Conference on Program Comprehension. ICPC, IEEE, pp. 335–346.

Pollock, L., Vijay-Shanker, K., Shepherd, D., Hill, E., Fry, Z.P., Maloor, K., 2007. Introducing natural language program analysis. In: Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. ACM, pp. 15–16.

Sridhara, G., Hill, E., Muppaneni, D., Pollock, L., Vijay-Shanker, K., 2010. Towards automatically generating summary comments for java methods. In: Proceedings of the International Conference on Automated Software Engineering. ACM, Antwerp, Belgium, pp. 43–52.

Taneja, K., Xie, T., 2008. DiffGen: Automated regression unit-test generation. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering. IEEE, L'Aquila, Italy, pp. 407–410.

Thummalapenta, S., Xie, T., Tillmann, N., De Halleux, J., Schulte, W., 2009. MSeqGen: Object-oriented unit-test generation via mining source code. In: Proceedings of the Joint Meeting of the European Software Engineering Conference and the SIGSOFT Symposium on the Foundations of Software Engineering. ACM, Amsterdam, Netherlands, pp. 193–202.

Uri Alon, O.L., Yahav, E., 2022. Code2vec. Accessed: 2022-09-12, https://github.com/tech-srl/code2vec.

Wu, J., Clause, J., 2020. A pattern-based approach to detect and improve non-descriptive test names. J. Syst. Softw. 168, 110639.

Wu, J., Clause, J., 2022a. Automated identification of uniqueness in junit tests. ACM Trans. Softw. Eng. Methodol..

Wu, J., Clause, J., 2022b. Dataset for "Provide Developer-Approved Descriptive Names for Unit Tests". http://dx.doi.org/10.5281/zenodo.7305997.

Wu, J., Clause, J., 2023. Example Survey and Collection Survey Data for "A Uniqueness-based Approach to Provide Descriptive JUnit Test Names". http://dx.doi.org/10.5281/zenodo.7830453.

Yamanaka, J., Hayase, Y., Amagasa, T., 2021. Recommending extract method refactoring based on confidence of predicted method name. arXiv preprint arXiv:2108.11011.

Zhang, B., Hill, E., Clause, J., 2015. Automatically generating test templates from test names. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, Lincoln, NE, USA, pp. 506–511.

Zhang, B., Hill, E., Clause, J., 2016. Towards automatically generating descriptive names for unit tests. In: Proceedings of the International Conference on Automated Software Engineering. ACM, Singapore, Singapore, pp. 625–636.

Zhang, S., Jalali, D., Wuttke, J., Muşlu, K., Lam, W., Ernst, M.D., Notkin, D., 2014. Empirically revisiting the test independence assumption. In: Proceedings of the International Symposium on Software Testing and Analysis. ACM, San Jose, CA, USA, pp. 385–396.

Zhou, Y., Yan, X., Yang, W., Chen, T., Huang, Z., 2019. Augmenting java method comments generation with context information based on neural networks. J. Syst. Softw. 156, 328–340.

Jianwei Wu is a Senior Software Engineer in Test and staff member of the Quality Engineering team at MathWorks. He received his Ph.D. degree in computer science from the University of Delaware. He worked as a research assistant at the University of Delaware under the advisement of Dr. James Clause. The main interests of his research are software testing and documentation.

James Clause is an Associate Professor in the Department of Computer and Information Sciences at the University of Delaware. He received the MS and Ph.D. degrees in computer science from the University of Pittsburgh and the Georgia Institute of Technology, respectively. His primary areas of research are software testing, program analysis, green software engineering, and documentation.