Empirical Software Engineering 2025 30:126

Supporting Multi-dimensional Unit Test Classification

Ziming Wang · Martin P. Robillard

Accepted: 14 April 2025

Abstract In software development projects, unit test names contribute to the overall quality of the tests. Developers often encode contextual information in the test names to enhance the readability and maintainability of tests. However, this information lacks a formal structure, and thus cannot be systematically used to support software development practices such as documentation and test refactoring. Additionally, large test suites can remain hard to read and maintain, even with descriptive test names. To address these limitations, we propose to identify common types of information encoded in test names using prevalent test naming conventions. We introduce a novel rule-based approach, called Sift4J, to automatically extract latent semantic information encoded in the name of a unit test. Information fragments we can extract from test names include the name of the method under test, a description of the state of the object under test, and the expected result of executing the unit under test. We demonstrate how to perform multi-dimensional classification of unit tests using this information. We evaluate the performance of Sift4J on two samples of unit tests: our development set and a previously-unseen evaluation benchmark. The results show that we can extract sufficient information from test names to assist in meaningfully reorganizing the tests in test classes.

Keywords Software Testing · Test Maintenance · Refactoring · Test Structure

Z. Wang
School of Computer Science
McGill University
Montréal, QC, CA
E-mail: ziming.wang2@mail.mcgill.ca

M. P. Robillard
School of Computer Science
McGill University
Montréal, QC, CA
E-mail: robillard@acm.org

1 Introduction

Readability and maintainability are key quality attributes for unit tests [7]. Test method names often have an impact on test suite readability and maintainability [33], as they are one immediate source of information for understanding the intent of test suites. Developers can benefit in multiple ways from descriptive names. For example, descriptive names can help developers understand the intent of the unit test without reading the test body [5]. Thus, developers often encode semantic information in the test names (e.g., the name of the unit under test, the feature under the test, or the expected outcome of the test). However, the encoding of information along these different *dimensions* is not checked by the compiler, and can thus be prone to inconsistencies [27] and be difficult to use by tools. In addition, long test suites can remain hard to read and maintain even with descriptive test names. To help overcome these problems, we investigate what important information developers commonly include in test method names, to what extent this information can be automatically identified, and how this information can help organize a test suite. Our investigation is conducted in the context of software development in Java using the JUnit testing framework.

We first identify common types of information encoded in test names by eliciting prevalent test naming conventions. Based on these findings, we propose a novel rule-based approach, called Sift4J, for extracting information fragments from Java unit tests and encoding this information explicitly as Java annotations. Sift4J comprises a collection of semantic fragment extraction rules, each of which is associated with a family of test naming conventions. Sift4J uses an ensemble of information extraction techniques that include textual analysis using regular expressions, static analysis of the test code, and natural language processing to convert the information in test names to Java annotations. The benefits of converting informal metadata about tests into annotations are manifold: they provide a statically-checked consistent information schema (the annotation types), and support improved analysis and manipulation of the tests by annotations processors. As a proof of concept of the usability of the information we can extract from tests, we developed an IntelliJ plug-in to allow users to browse and organize the tests in a test class according to the various *dimensions* determined by the information fragments in test names (e.g., to group them by the focal method targeted by each test).

We evaluated Sift4J by measuring its accuracy on two samples of unit tests: our development set and a previously-unseen evaluation benchmark of Java unit tests that use JUnit framework. The results show that we can extract sufficient information from test names to assist in meaningfully reorganizing the tests in test classes. This work makes the following contributions:

1. A general and language-independent formulation of the problem of semantic information fragment detection in a unit test name;
2. A catalog of naming conventions and their corresponding semantic information fragments types identified from a sample of Java unit tests;
3. A benchmark of unit test names and their naming conventions;
4. The design of a tool to automatically extract the semantic information fragments from Java unit tests that use the JUnit framework, and a proof-of-

concept plug-in that performs multi-dimensional classification on the Java tests annotated by Sift4J;
5. Empirical data evaluating the performance of Sift4J tool for extracting information from tests.

The remainder of this paper is structured as follows. Section 2 discusses relevant past research and presents a precise formulation of the semantic information fragment detection problem. Section 3 details a formative study of test name conventions, including its methodology, and presents the resulting types of semantic information fragments and prevalent naming conventions we identified. Section 4 describes the design of the Sift4J tool for extracting information fragments from test names, including a number of information extraction techniques and a discussion of its limitations. It also presents how multi-dimensional test classification can be achieved within an integrated development environment (IDE). Section 5 presents the design and outcome of an evaluation study, followed by a conclusion in Section 6. This article is complemented by a online dataset [31].

2 Information Fragments in Test Names

This research is predicated on the observation that the names of unit tests commonly encode information about different properties of the test, and this information may be systematically organized through a naming convention. For example, a test named testIsHorizontal_False for a class representing a geometric line encodes two pieces of information about the test: the name of the method being tested (isHorizontal), and the expected outcome of the evaluation of this unit under test (in this case, a return value of false). In this example, the information fragments are made prominent with the help of two syntactic features: a *test prefix marker* (test), and a *separator* (_), and the applied naming convention can be expressed as test[FocalMethod]_[ExpectedResult].

We henceforth refer to a cohesive piece of information about a unit test as a *semantic information fragment* (or simply, *fragment*). We hypothesize that fragments can be extracted from the names of unit tests with the help of naming conventions. As this work is scoped in the context of the Java language, we consider that a unit test corresponds to a test method as identified by the JUnit framework, and that the *name* of the test is simply the test method's simple name. A test name can be *tokenized* into a sequence of tokens based on lexical or syntactic features, such as case or the use of separators. The example above would be tokenized as test,Is, Horizontal, _, False.

2.1 Related Work

There is ample evidence that developers informally encode semantic information as fragments in unit test names. This evidence can be found both in the grey and the scientific literature, and is easily confirmed by inspection of test suites (see Section 3). In terms of grey literature, numerous blogs mention conventions for naming unit tests that involve different kinds of semantic encoding (e.g., [10,11,13, 20,29]). A common piece of advice is to encode the name of the unit under test (or

focal method [6]) in the test name. Another common recommendation is to include a description of the expected behavior of the unit under test (same references). There is currently no common standard for structuring this information in tests, and practices vary. Some conventions require prefix markers (typically `test`), while some omit this marker. Likewise, token separation can be done using different lexical features (e.g., CamelCase or snake_case), or explicit tokens such as `should` and `when`, or any combinations of the various possible alternatives.

Previous research also provides, directly or indirectly, useful insights about the kinds of information that is or should be part of a test name. *Test-to-code traceability techniques* aim to discover the link between test code and the code being tested (e.g., [6, 22, 24–26, 28]). The motivation for this research is that this link, useful for various test suite maintenance activities, can be lost if it is not documented. Explicitly providing the name of the focal method in the test name thereby helps avoid the cost of recovering this link. Ghafari et al.'s work, in particular, focused on recovering focal methods using data-flow analysis [6].

Past work has also addressed the challenges of *automatically generating names for unit tests*, or *test templates from test names*. From these efforts, we can learn about properties of the information that is recommended to be present in test names by the designers of the various approaches. Zhang et al. proposed to leverage information in test names to generate an implementation template for the test [37]. Their proposal relies on the assumption that the test name would follow a "well-defined grammatical structure" that consists of an "action phrase" followed by a "predicate phrase", both expressed as verb phrases. In later work, the same research group proposed a technique to go the other way, and automatically generate a test's name that "summarizes the test's scenario and the expected outcome" [38]. Similarly, Daka et al. proposed a technique to generate names that follow a three-part naming convention to generate descriptive test method names, including the method under test, the state under test, and the expected behavior [5]. Wu and Clause devised a pattern-based approach to compare test names and their corresponding bodies [34]. In doing so, they also considered three types of information from both the test method name and body: action, predicate, and scenario. Wu and Clause further leveraged this information and proposed a uniqueness-based approach to generate test names [35, 36]. Another interesting approach was introduced by Allamanis et al. to predict the test name from the test body using a neural probabilistic language model [1].

In addition, Peruma et al. used grammatical patterns to interpret test names for the purpose of supporting their evolution [23]. As part of this work, they observed an impressive variety of ways to express test information in test names. The previous work has shown that descriptive test method names are an asset for improving the quality of unit tests, and that it is reasonable to expect that tests can follow some naming convention. However, we found that there is no agreement on what information should be included in test method names and, more importantly, there is no uniform way to express this information.

Finally, previous research has also provided indirect insights on how to manage large test suites. Greiler et al. showed that the low cohesive test methods grouped in the same class may result in test smells [8]. Kochhar et al. [12] conducted open-ended interviews to identify 29 hypotheses that describe characteristics of good test cases, and surveyed 261 practitioners to validate these hypotheses. Most of their respondents agree that large test cases are hard to understand and maintain,

and the use of tags or categories is helpful to manage test suites in practice, for example, for running a specific set of tests at a time. JUnit provides a set of annotations to tag test cases. In particular the `@Nested` and `@Tag` annotations can be used to help organize test suites. The `@Nested` annotation can be used to further group test methods inside nested classes. The `@Tag` annotation is intended to declare a tag for the corresponding test classes or methods, which can be then be used to filter which tests are executed. Another related research is from Li et al. [15], who predefined a catalog of 21 stereotypes, which are mostly JUnit API-based. They also developed a tool to automatically generate the stereotypes from the test methods and tag the tests with the generated stereotypes, which can assist navigation/classification of a group of tests.

2.2 Problem Formulation

If we accept that a test name is likely to follow a naming convention containing information about the test, we wish to extract this information from the name by leveraging the naming convention. We define the problem of *extracting semantic information fragments from test names* (*fragment extraction* for short) as a function that takes as input a test name and its context, and produces a sequence of *tagged fragment* tuples $\{(F_1, T_1), ...(F_n, T_n)\}$. In a tuple (F, T), F is a substring of the test name and T is a configurable tag that describes the nature of the fragment. In practice, the *context* for a test name is the code base that contains the test together with its necessary dependencies. A sequence of tagged fragments can be characterized by the types of fragments it tags.

Returning to our example above, one solution to the fragment extraction of `testIsHorizontal_False` could be, in a given context:

$$\textsc{Method–Result}\text{: } \{(\mathsf{isHorizontal}, \textsc{Method}), (\mathsf{False}, \textsc{Result})\}.$$

Designing a technique to solve the fragment extraction problem requires a precise understanding of the types of fragments that it is possible to encounter in practice. We conducted a formative study to elicit these types.

3 Types of Semantic Information Fragments

We conducted a formative study to answer the questions *what types of semantic fragments can we find in unit tests written in Java? How do they manifest?* The answers to these questions provide a framework for extracting semantic information fragments in unit tests based on current practice. The study consisted in assembling a diverse sample of unit tests, then inspecting each test in context and manually classifying the information fragments in its name using a qualitative coding process. The *context* for a unit test name includes the source code of the test suite, including the test itself, which we leveraged for the classification.

3.1 Methodology

We used GitHub Search and the GitHub Search API[1] to collect 100 public repositories with Java test code. We considered a repository eligible if it was tagged by GitHub as containing Java code, and if it contained at least 50 test files. We define a *test file* as any file that 1) has the .java extension and 2) contains the string test in its path, and 3) uses the JUnit framework.[2] We conducted the query on 27 November 2022 and selected the 100 most-starred repositories that met these inclusion criteria.[3]

Next, we sampled unit tests from the 100 repositories with the goal of recording as many different test name structures as possible for a reasonable manual inspection effort. For this purpose, we randomly sampled one test class per repository, and inspected all its test methods as identified with the @Test annotation. For each test, we assigned a *label* to describe the *naming convention* used for the test. We then repeated the entire process until we reached saturation, which we defined as inspecting 20 consecutive test classes without encountering a new naming convention. We reached saturation after three iterations, thereby collecting data on 1263 test methods from 300 classes.[4] Of these methods, 18 had names that clearly captured no information about the test (e.g., methods named simply test, or test1). We discarded these methods from further analysis, leaving us with a data set of 1245 unit tests. We then collapsed the set of naming conventions into a set of *convention families*, each capturing a different sequence of information fragments about a test.

Eliciting Naming Convention We labeled each test using a combination of *keywords*, *separators* and *placeholders* to represent a naming convention. For example, we would assign the label test[Focal Method]_[Expected Result] to the method testIsHorizontal_False. We derived the labels describing each naming convention using a manual inspection process informed by the grey literature on naming conventions for unit tests (see Section 2.1). In a test name, keywords and separators can be readily identified by recognizing substrings such as test or when. Identifying instances of placeholders is a more important task as its outcome determines the types of information fragments we can detect from test names. For this purpose we considered different groups of tokens in the test name and attempted to match them with common testing concepts discussed in the grey literature, creating new types of placeholders as necessary. A single investigator conducted this analysis.

Defining Convention Families Our focus is on the type of information we can extract from tests. To pave over insignificant differences in encoding style, we analyze our findings in terms of *naming convention families*. We group naming conventions together in a family if they differ only in terms of delimitation style (e.g., camelCamel case vs. snake case) and/or choice of explicit token (e.g., test,

[1] github.com/search and docs.github.com/rest, resp.

[2] We used the GitHub API to check if test files contained the string junit.

[3] In practice, we retrieved the 300 most-starred Java repositories and analyzed each in decreasing order of stars until we collected 100 with testing code.

[4] When repeating the process, we ensured that any test class selected from a previously-sampled repository was located in a different package from any of the test classes previously sampled from this repository.

return, with). For example, we grouped the conventions [Method]Test and test_[Method] together in the Method Only family. Finally, given a convention family, we can trivially extract all the fragment types used as placeholders. For example, from the convention family Method–Result we extract the information fragments (Focal) Method and (Expected) Result.

3.2 Results

Table 1 lists the fragment types we cataloged, together with statistics of their observation frequency in our data set of 1245 test methods. The third column (*Obs.*) provides the number of tests whose name included a semantic fragment of the corresponding type. The fourth column (*Prop.*) divides this number by 1245 to provide a ratio. The sum of ratios exceeds 100% because test names can include multiple fragments.

Table 1 Types of Semantic Information Fragments Observed in a Sample of Java Unit Tests

Fragment Type	**Description**	**Obs.**	**Prop.**
Method	Refers to the method under test. The method should be called within the test.	464	37%
Abbreviated Method	Refers to a subset of the tokens that form the name of the focal method.	96	8%
Class	Refers to the class under test.	82	7%
State	Refers to input state related to Focal Method	630	51%
Result	Refers to the expected outcome of the test case, including exception.	428	34%
Scenario	A general description of the focus of the test when no category applies that would be more specific.	225	18%

Table 2 lists the convention families we observed, with their frequency. Eighteen types of convention with at least ten instances cover 96% of our sample test (1195/1245). The Method Only family is the most prevalent, constituting 16% of the observations. These observations show that the vast majority of test names encode at least one semantic information fragment.

As expected, the main practices we detected involve specifying the name of the focal method (37%). This practice also has the advantage of being *unambiguous*. Except when testing overloaded or overridden methods accessed polymorphically, it can be possible to refer to precisely the method under test. To a certain extent, precise references are also possible for values of variables and arguments. Unfortunately, the same cannot be said of vaguer concepts such as State or Scenario. Our research explores how to resolve ambiguous references to this kind of semantic information through a heuristic approach that combines regular expression matching, static analysis, and natural language processing.

Table 2 Naming Convention Families Observed in a Sample of 1245 Java Unit Tests

Convention Family	Frequency
Method Only	204
Method–State	136
Result Only	134
State Only	123
Scenario Only	123
Result–State	113
Method–State–Result	49
Abbreviated Method Only	47
Class Only	46
Abbreviated Method–State	44
Scenario–State	40
State–Result	35
Class–State	24
Method–Result	21
Scenario–Result	18
State–Scenario–Result	14
Scenario–State–Result	12
Result–Method–State	12
Method–Result–State, Method–Method	7
Result–Scenario	6
Method–Class	4
State–Scenario, Scenario–Class	3
Class–Method–Method, Abbreviated Method–State–Result, Method–State–Method, State–Abbreviated Method–Result, Method–Method–State, Method–State–State	2
Method–State–Result–State, Class–Method, Scenario–State–State, Scenario–Abbreviated Method, Method–State–Scenario–Result, Class–Scenario, Scenario–Class–Result, Scenario–Result–State	1

3.3 Limitations

A main limitation of the study is that the sample is not uniformly random and therefore cannot support the inference of fragment type proportions to a broader population of unit tests. However, such inference was not the goal of study. The differences in proportions we observe are sufficiently distinct to help us prioritize the development of basic classification rules. For example, having observed 464 instances of unit tests that name the focal method in the test name in some of the most popular Java projects on GitHub, we have confidence that we are not attempting to support an exotic practice. The second limitation concerns the accuracy of the manual classification. Classifying fragment types according to the protocol described above amounts to a program understanding task, which can leave some room for personal interpretation. We deemed it unnecessary to employ two coders to calculate inter-rater reliability for this task for two reasons. First, it is a low-subjectivity task as many placeholders map directly to program constructs (e.g., focal method, parameter name). Second, minor mischaracterizations have limited practical impact as we are primarily interested in the diversity of information types as opposed to the precise frequency of their occurrence in our data set. Our data set is also available for independent verification.

```
1  @Test
2  @FocalMethod(''IsEmpty'')
3  @State(''CollectionIsEmpty'')
4  @ExpectedResult(''ReturnTrue'')
5  public void testIsEmpty_whenCollectionIsEmpty_thenReturnTrue() {
6   Collection<Object> testCollection = new ArrayList<>();
7   assertTrue(''Should return true because collection is empty'',
8     CollectionUtils.isEmpty(testCollection));
9  }
```

Fig. 1 A Sample Unit Test with Annotated Semantic Information Fragments

4 Extracting Semantic Information From Tests

We contribute the design of a technique for extracting information fragments from unit tests, as formulated in Section 2.2. We implemented a prototype for Java we call *Sift4J* (for *Semantic Information From Tests for Java*). Sift4J serves as a proof of concept of the feasibility of extracting information fragments from Java unit tests. The prototype is structured as a rule engine with a collection of semantic fragment *extraction rules* applied sequentially to a unit test. Each rule is associated with a naming convention family as identified in Table 2. The input to Sift4J is a test file and associated code base. The output is an updated version of the input test file with Java annotations indicating any detected information fragment. The listing of Figure 1 shows an example of unit test annotated with Sift4J. We developed the Sift4J prototype through iterative design informed by a *development set* consisting of a sample of 442 units tests. The composition of this development set is detailed in Section 5.

4.1 Overall Architecture

The Sift4J rule engine is implemented in Java and operates by parsing an input Java source file containing unit tests, and then providing these tests to a number of extraction rules. Figure 2 provides a view of the essential elements of the Sift4J design.

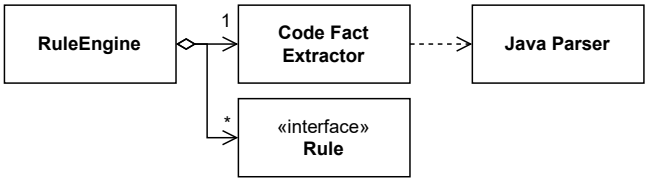

Fig. 2 Sift4J Overall Design

The RuleEngine relies on a CodeFactExtractor to obtain the list of unit tests for a Java source file. These unit tests are returned in the form of a MethodDeclaration Abstract Syntax Tree (AST) node. The CodeFactExtractor relies on the JavaParser

library to parse source files and resolve as many of the symbols therein as possible.[5] The RuleEngine class can be configured with any number of instances of type Rule. An instance of Rule provides the computation necessary to detect a naming convention from a test's name according to a given heuristic (e.g., by linking text in the method name to a production focal method). An instance of type Rule is employed by calling an apply method with a method declaration node representing a unit test as input. Applying a rule returns an optionally empty *convention* encoding six potentially-empty string instances representing the types of semantic fragments we identified in our formative study (see Section 2.2). The returned convention is empty if the rule does not match the input test. If the application of a rule successfully detects a convention and returns a non-empty result, the remaining rules are short-circuited (i.e., not executed). In our design, a rule falls into one of two categories: single-fragment or multiple-fragment. These categories are represented as abstract classes in our design. A rule is defined by extending the corresponding class, instantiating it, and adding the instance to the rule engine's list of rules.

We designed and implemented a number of *predefined* extraction rules to demonstrate the approach and support experimentation. As targets for our predefined rules, we chose to implement support for all convention families for which we had observed over ten instances in our formative study (see Table 2). However, in the list of 18 target conventions, two were not amenable to automatic detection via heuristics: Scenario Only, and Scenario–Result. Because of a lack of structure and constraints for expressing scenarios described in free-form text (e.g., testCreatingStreamAfterClose), there is no explicit feature we could rely on to design extraction rules for these families. We implemented support for extracting information fragments for all 16 remaining convention families. These predefined rules are not intended to cover all conventions potentially in use, but they enable our further empirical investigation. To support the pragmatic eventuality that some projects may use idiosyncratic conventions to name their unit tests, we engineered our solution as a flexible framework that allows users to define an open-ended collection of arbitrary *custom rules*.

4.2 Extraction Techniques

We designed Sift4J's predefined rules using a combination of four *extraction techniques*: 1) Matching against well-defined conventions encoded as regular expressions, which we call *convention patterns*; 2) Using *static analysis* to match the test name to code elements from the body of the test; 3) Using natural language processing to infer fragment types from parts of speech (e.g., *noun phrase*), which we refer to as *grammatical patterns*; 4) Detecting fragments from the presence of selected *keywords* at specific positions in the test name. Table 3 reports the subset of techniques we employ for each rule.

Convention Patterns A number of test naming conventions use a well-defined and unambiguous pattern than can be readily detected, e.g., the convention given[State]_then[Result]. We refer to such practices as *convention patterns*. We use

[5] We used JavaParser version 3.25.1 configured with a symbol solver that combines the JavaParserTypeSolver and the ReflectionTypeSolver.

Table 3 Extraction Techniques Applied in Predefined Rule Set. The convention types are listed in the same order as they are applied by Sift4J.

Convention Type	Convention Pattern	Static Analysis	Gramm. Relations	Keywords
Class Only		✓		
Method Only		✓		
Abbreviated Method Only		✓		
Result–Method–State	✓	✓		
Method–State–Result	✓	✓		
State–Scenario–Result	✓			
Scenario–State–Result	✓			
Method–State	✓	✓	✓	✓
Class–State		✓	✓	✓
Method–Result		✓	✓	✓
Abbreviated Method–State	✓	✓	✓	✓
Result–State	✓	✓	✓	✓
State–Result	✓	✓	✓	✓
Scenario–State	✓		✓	
State Only		✓		✓
Result Only	✓	✓		✓

a regular expression to detect instances of the convention and extract the corresponding fragments. In our example, the instance of the convention can be detected with the regular expression given(\w+)_then(\w+) as part of executing the State–Result rule. For example, for the input test givenEmptySets_thenExpectNoChanges, the resulting sequence of semantic fragments is extracted as {(EmptySets: State), (ExpectNoChanges: Result)}. We encoded 53 such patterns elicited from our development set. The majority (25/53) start the test name with either should or testShould, e.g., should_[Result]_when_[State]. This extraction technique is the only one capable of extracting fragments of type Scenario, because in some naming conventions the scenario is a placeholder in a well-defined pattern, e.g., [Scenario]_with[State]_returns[Result].

Static Analysis We use static analysis to link the text in the test method name to the program entities in the test. The static analysis strategy depends on the type of semantic fragments to be extracted. For example, a test named testGetResources can be linked to a focal method getResources if a call to such a method can be detected in the body of the test. Table 4 provides additional details, and Section 5.3.1 reports on the sensitivity analysis we conducted to determine the similarity thresholds.

Grammatical Patterns We observed in the formative study that certain grammatical structures can be indicative of the presence of a specific type of semantic information. For example, a prepositional phrase (e.g., withNull), appearing after a method name is likely to describe the input State of the focal method (e.g., test_isHorizontal_withNull). Table 5 documents the grammatical patterns we observed and leverage. We use the part-of-speech (POS) tagger of the Stanford Core NLP library [17] to perform the grammatical structure analysis.

Table 4 Static Analysis Strategies for Extracting Semantic Fragments

Fragment Type	Extraction Technique	Tagged Text
Method / Abbreviated Method	Combine independent heuristics to compute a score following the strategy of White et al. [32]. Obtain the name of all methods called directly in the test, compute four case-insensitive similarity measures between the name of the method called and test name, and add the results. The similarity measures are: exact match, exact containment, Levenshtein distance, and longest common subsequence.	The (partial) name of the called method identified as similar to the test name.
Class	Use the same approach as above. Instead of collecting method calls, we collect classes of the objects created as well as the classes passed to the focal method as method arguments within the test body.	The name of the class identified as similar to the test name.
State	Generate a state description based on the API-Coverage goal following the strategy of Daka [5]. Obtain the names and values of all arguments declared in the test method and, if the name is longer than one character, check if they are contained in the unit test name. If not, generate a description to describe the collected arguments based on their type and quantity, and identify if the description is similar to part of the test name.	The name of the argument or the generated state description identified.
Result	Generate a result description based on the assert statement type following the strategy of Zhang [38]. Obtain the last assert statement in the test body, generate a description based on the assertion type and the arguments passed to the assert statement, and identify if the description is similar to part of the test name. For testing exceptions, we attempt three strategies: 1. Use the `expected` attribute of JUnit's `@Test` annotation 2. Detect the try–catch idiom 3. Analyze calls to JUnit5's `assertThrows` method.	The generated result description or the name of the exception identified.

Keywords We also leverage the simple heuristic that certain common terms in a method name can indicate the presence of specific types of information fragments [23]. For example, the terms `empty`, `single`, `double` are likely to describe the quantity of the input passed to the method under test, implying that the fragment is State. We seeded a glossary of keywords based on the cues present in our development set. The glossary consists of a set of keywords associated with one of two positions in the test name: starting, or within. Each keyword is then associated with a fragment type (either State or Result). As this technique is used as a last resort (see Section 4.3), we purposefully kept the glossary minimal.[6] In practice, this glossary could be tailored to specific projects for best performance.

[6] For example, we only defined 18 starting position keywords associated with the type State. These include: `single`, `double`, `multi`, `empty`, `invalid`, `null`, and `default`.

Table 5 Grammatical Patterns for Semantic Information Fragments. In the column *Pattern*, parts of speech are abbreviated as follows: NP: Noun Phrase; PP: Prepositional Phrase; ADJP: Adjectival Phrase; VP: Verb Phrase

Rule	Pattern	Example
Method–State	Method + NP	edgesConnecting_disconnectedNodes (edgesConnecting: Method)
	Method + PP	retryTimesPredicateWithZeroRetries (retryTimesPredicate: Method)
	Method + ADJP	testGetInReplyTo_empty (GetInReplyTo: Method)
Abbreviated Method–State	Abbreviated Method + NP	testGetRep_1bytechar (GetRep: Abbreviated Method)
	Abbreviated Method + PP	testFitForEqualProcecesses (Fit: Abbreviated Method)
	Abbreviated Method + ADJP	testResourceSetsNull (ResourceSets: Abbreviated Method)
Class–State	Class + NP	namespace_invalidFormat (namespace: Class)
	Class + PP	testLogicDeleteByPrimaryKey (LogicDelete: Class)
	Class + ADJP	asyncExpiry_pending (asyncExpiry: Class)
Method–Result	Method + VP	isValid_shouldValidateConfigRepo (isValid: Method)
State–Result	State + VP	aUUIDStringReturnsAUUIDObject (aUUIDString: State)
Result–State	VP + State	shouldIncreaseCounterWithNegativeValues (WithNegativeValues: State)
Scenario–State	NP + State	cycleOfMixedWithImmutableRoot (WithImmutableRoot: State)

4.3 Rule Implementation

Our implementation strategy for predefined rules follows an opportunistic approach with fallbacks. In other words, we try to detect if a test name matches an extraction rule by using the most precise technique first (i.e., convention patterns), and then falling back onto progressively less precise alternatives as necessary. In the case of extraction rules for multiple fragments, some rules can take into account the partial matching of the test's name by one technique when applying other techniques. For this reason, the rule meta-heuristic differs slightly for rules to extract a single fragment (Algorithm 1) from rules to extract more than one fragment (Algorithm 2).

Our use of a combination of alternative extraction techniques for information fragments implies the possibility that more than one rule can match an input test. To address this eventuality, we order the rules by decreasing expected likelihood of a correct match. The specific strategy we used is to first apply rules based only on static analysis, since they are the most precise. Then we order rules in descending number of fragments they detect, as detecting each fragment involves additional checks that can reject a spurious match. Table 3 shows the rule execution order we selected. Once a rule successfully detects a convention, the remaining rules are skipped.

```
1  public void testIpAddress_whenInitializedFalse() {
2   service.ipAddress(IP_ADDRESS);
3   String ipAddress = Whitebox.getInternalState(service, ''ipAddress'');
4   assertEquals(''IP address should be set to the IP address that was specified'',
        IP_ADDRESS, ipAddress);
5  }
```

Fig. 3 A Sample Unit Test Employing the Method–State Naming Convention

We illustrate how the rules are applied and executed by walking through a test from test class ServiceTest of the Spark project (perwendel/spark, see complete details in our online dataset [31]). Figure 3 shows the code of the unit test from which we wish to extract information fragments. This test employs the Method–State naming convention because it consists of the keyword test, followed by the name of the method under test (ipAddress), and a description of the state under test (whenInitializedFalse).

Once Sift4J has instantiated the rule engine and parsed the code, it cycles through the rules one by one. The first rule it considers is Class Only (see Table 3). As this is a *single fragment rule*, its implementation follows Algorithm 1. First, it checks if the name matches a *convention pattern* (lines 2–5), which it does not. Second, the string is *preprocessed* to convert it to the normalized form ipAddressWhenInitializedFalse (line 6). The execution then attempts to recover the class name using *static analysis* (line 7), which fails because the preprocessed name does not contain the name of the test class. Finally, since for the Class Only rule there are no applicable keywords (line 10), the rule fails (line 13) and the rule engine cycles to the next rule.

Algorithm 1 One-Fragment Convention Rule Extraction Algorithm

Input: U: Unit Test Declaration
Output: C: a Convention Instance

```
1: n ← Unit Test Name
2: if n follows a Convention Pattern then
3:     f ← applyRegularExpression(n)
4:     return buildConvention(f)
5: end if
6: n ← preprocess(n)                    ▷ Remove underscores, "test", and related tokens
7: f ← performStaticAnalysis(U)
8: if f matches n then
9:     return buildConvention(f)
10: else if n starts with Special Term then
11:     return buildConvention(f)
12: end if
13: return Empty
```

After cycling through subsequent rules, which all fail because the test name does not match their requirements, the rule engine reaches rule Method–State. This is a *multiple-fragment rule*, whose implementation follows Algorithm 2. The first part of the execution is similar to Algorithm 1: the execution fails to detect a convention pattern, and normalizes the test name (lines 2–6). Execution moves to the static analysis strategy, which attempts to produce two fragments from the

code of the test (line 7). The first fragment is the name of a method called within the body of the test which is also contained in the name of the test, in our case, ipAddress. In the implementation of the METHOD–STATE rule, we attempt to create a second fragment by checking if the name of the parameter or argument values are contained in the remainder of the test name, as described in Table 4. Interestingly, here the heuristic produces a false positive because it detects IP_ADDRESS, the parameter to ipAddress as a description of the state. However, the rule's implementation is robust to this imprecision because the test name does not end with the spurious state fragment ipAddress (line 8), so the execution attempts to match the condition on line 11, which it does because the normalized name starts with ipAddress, producing a remainder of WhenInitializedFalse (line 13). This remainder is then checked for grammatical relations or keywords. Because the keyword When is associated with a state fragment, the rule returns the convention: METHOD–STATE: {(ipAddress, METHOD), (WhenInitializedFalse, STATE)} and short-circuits applying the remainder of the rules.

Algorithm 2 Two-Fragments Convention Rule Extraction Algorithm

Input: U: Unit Test Declaration
Output: C: a Convention Instance

1: $n \leftarrow$ *Unit Test Name*
2: **if** n follows a *Convention Pattern* **then**
3: $f1, f2 \leftarrow$ applyRegularExpression(n)
4: **return** buildConvention(f1, f2)
5: **end if**
6: $n \leftarrow$ preprocess(n) ▷ Remove underscores, "test", and related tokens
7: $f1, f2 \leftarrow$ performStaticAnalysis(U)
8: **if** n starts with $f1 \wedge$ ends with $f2$ **then**
9: **return** buildConvention(f1, f2)
10: **end if**
11: **if** n starts with $f1 \vee$ ends with $f2$ **then**
12: $f \leftarrow$ the matched fragment
13: *remain* $\leftarrow$ remove f from n
14: **if** *remain* follows *Grammatical Relation* $\vee$ starts with *Special Term* **then**
15: **return** buildConvention(f, remain)
16: **end if**
17: **end if**
18: **return** Empty

4.4 Limitations of Rule-Based Semantic Fragment Extraction

We opted for a rule-base approach to provide a direct traceability between information fragments and source code. In addition to providing a clear rationale for the detection of a fragment through the rule family employed to detect it, the use of a rule-based approach provides clear guidance for developers wishing to encode semantic fragments in their test name. The limitations of Sift4J are thus a manifestation of the fundamental limitations of rule-based systems applied to our context. First, not all information can be encoded by following conventions. Second, a heuristic approach to match natural language is imprecise and incom-

```
1  @Test
2  public void testFloorDoubleNumber()
3  {
4   assertEquals(0, Floor.floor(0.1));
5   assertEquals(1, Floor.floor(1.9));
6   assertEquals(−2, Floor.floor(−1.1));
7   assertEquals(−43, Floor.floor(−42.7));
8  }
```

Fig. 4 Example of reuse term in information semantic fragment.

plete by nature. Third, the performance of the approach is impacted by technical aspects of the extraction techniques.

The first limitation is a reflection that test names are often in free-form natural language that does not follow any detectable convention. In our framework, this situation is explicitly captured by convention families with potentially unspecified fragments, such as State, Result, or Scenario (see Table 2). In cases where developers use free-form text to describe a scenario that involves an arbitrary collection of code elements, there is no clear traceability principle that can be used to identify semantic fragments. For example, if a test is named sanity to indicate that the test case is validating the basic functionality for a method, Sift4J will be unable to extract an information fragment from the name. This limitation is compounded by the fact that, even when a project uses a well-defined test naming convention, it is possible that not all test names consistently follow that convention. Consistency is especially impacted by the challenges of co-evolving test and code [27,30]. For example, if a production method named getParams is renamed params, but the corresponding test testGetParams is not updated accordingly, Sift4J will not detect an instance of the Method Only convention.

A second limitation is that, because test names do not have to follow a formal structure checked by the compiler, ambiguities can occur, or the heuristic rules can be insufficiently precise to detect the encoded information. An example of ambiguity is a test named maxDelayIsNotMissedTooMuch making a call to a production method named is. In this case, Sift4J will falsely identify is as the focal method. Another example we have seen in practice is of a test named testFloorDoubleNumber, whose focal class and focal method are both named Floor (see Figure 4).

The third limitation is that the implementation of all four of our extraction techniques (Section 4.2) impacts the performance of the approach. For *Convention Patterns*, the implementation needs to include patterns used in a project for the approach to be applicable. Similarly, the *Keywords* approach will be sensitive to the glossary used as hints that certain tokens represent certain types of fragments, and these can vary from project to project. The *static analysis* technique relies on the correct parsing and type resolution of incomplete Java source code, which is itself an approximate process. We rely on JavaParser's built-in JavaSymbolSolver to resolve overloaded method calls. However, this implementation has its own limitations and bugs. As for matching the names of detected methods to the test name, we rely on a threshold value. We conducted a sensitivity analysis to gain confidence that our choice of threshold was sound (see Section 5.3.1), but operating the tool in a significantly different context may require a different threshold parameter. Finally, a word may have a different part-of-speech POS than usual in

a software-specific context [4, 9, 19], which could negatively impact the result of the *Grammatical Patterns* technique. However, the performance of the Stanford Part-of-Speech Tagger has previously been considered satisfactory on analyzing the grammar pattern of software identifiers [2, 23, 34]. Our primary means for mitigating the technical limitations of extraction techniques is our reliance on a fallback approach, wherein we systematically apply the most precise approach first and only rely on less precise alternatives when no other option succeeds.

4.5 Multi-dimensional Test Classification

Once unit tests are annotated with semantic information fragments (as illustrated in Figure 1), it becomes straightforward to use an annotation processor to reorganize a test file to group tests according to the different dimensions that correspond to the different information types. For example, a test class could be organized by focal test method, by common input states (e.g., an empty structure), or by expected result (e.g., all tests for conditions throwing exceptions).

As a proof of concept, we implemented a test organization tool as an IntelliJ plug-in we refer to as the *Sift4J plug-in*. The Sift4J plug-in allows a user to semi-automatically restructure a test file by leveraging the information fragments therein. By default, the plug-in groups the unit tests based on the most frequent semantic fragment type observed in the test file (e.g., Method). The plug-in also supports multiple levels of grouping (for example, first by Method, then State). Although multi-level grouping is likely excessive for small test classes, the feature allows exploring latent test suite design strategies for large test classes.

In addition to allowing developers to browse the tests in a class by different semantic groups, the plug-in also supports the option to encode a desired grouping in the test file. For this purpose we use the `@Nested` annotation provided by the JUnit5 framework. The `@Nested` annotation was originally designed to help organize tests into classes that can share the scaffolding available via an instance of their enclosing class. We additionally leverage this feature to signal that a group of unit tests shares the same semantic fragments, and thereby encode the relationship among several groups of tests.

We illustrate the workflow supported by the Sift4J plug-in with a walk-through of a relatively simple test file called `CollectionUtilsTest.java` This class contains tests of the miscellaneous collection utility methods. The test class contains six test cases. Conveniently, the test names consistently adhere to the Method–State–Result convention family. To automatically annotate tests with semantic fragment information, one would right-click on the target test class file in the IntelliJ project view and select the *Run Sift4J* command. The identified semantic fragments are then shown in a view provided by the plug-in and organized in a method × fragment table (see Figure 5). Because of consistent test naming, each test case in the input class happens to be correctly tagged by Sift4J. However, the use of the Sift4J plug-in is independent of the performance of the automated fragment extraction process. For imperfect fragment extraction outcomes, developers can adjust the fragment annotations by editing the test file as desired. It is also possible to envision adoption scenarios where fragments are manually created at test creation time, or the possibility of automatically injecting annotations using in-house tools (e.g., relying on traceability to test plans).

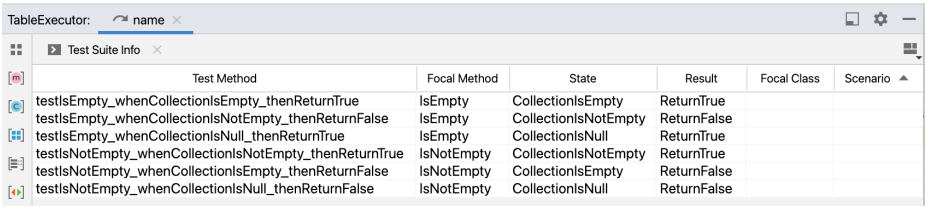

Fig. 5 View of the Sift4J plug-in. Buttons on the left side from top to bottom are: Classify by Default, Classify by Method Fragment, Classify by Class Fragment, Classify by State Fragment, Classify by Result Fragment, Classify by Scenario Fragment.

In any case, once tests are annotated with semantic information fragments, developers can use the plug-in to explore and/or refactor the test suite structure. Developers can select one of the classification strategies by clicking a corresponding button. Each classification strategy prioritizes grouping unit tests based on a different type of semantic fragment.

In the case of `CollectionUtilsTest`, for sake of illustration we refactor the test class first into two nested classes based on the Method information fragment type, as two focal methods are detected: `isEmpty` and `isNotEmpty`. Within each class, we further group the test cases based on the most frequent fragment value, excluding those already used. For the `isEmpty` class, two test cases shared the same Result fragment, `ReturnFalse`, so we generate a new nested class to group the tests accordingly, and follow a similar process for the `isNotEmpty` class.

5 Evaluation

Our goal was to evaluate Sift4J as a means to assess the feasibility of recovering semantic information fragments about unit tests in existing code. Once recovered, information fragments can be explicitly encoded via annotations, and thus provide long-term added value to the code base. However, multi-dimensional unit test classification is not a current practice and unit test naming conventions are neither standardized nor systematically followed in practice [27]. Hence, an estimate of the effort involved in recovering information fragments from code can guide adoption efforts. We designed a benchmark study to answer two research questions:

RQ1: How effective is Sift4J at correctly identifying conventions associated with predefined rules?
RQ2: For a correctly identified convention, how effective is Sift4J at extracting semantic information fragments encoded in test names?

5.1 Benchmarks

We developed and evaluated Sift4J by leveraging *benchmarks* of unit tests selected from existing Java projects. The design goals for our benchmarks were to collect *existing* tests from *recognized* code bases that exhibit *diverse naming conventions*. To meet these goals, we used a sampling strategy that combined purposive sampling of GitHub repositories with constrained random sampling within these repositories [3].

In developing the approach we leveraged a *development set* consisting of all the tests in 100 Java test classes randomly sampled from the 300 selected for our formative study (Section 3). For each test class included in the development set, we recorded the name of the selected repository, the name of the selected test class, and the names of all the test methods within the selected test class. For each test method, the first author manually determined the applicable convention family.

To evaluate the approach on unseen data, we created an *evaluation set* of 100 Java test classes by randomly selecting 100 additional test classes from the data collected in our formative study. We followed the same sampling procedure as described in Section 3.1, with an additional constraint that each test class should have at least ten test methods. We added this additional constraint because our multi-dimensional test classification approach targets test classes with many tests, as there is no point in spending effort organizing a class with only a handful of tests. Hence, selecting classes with a high number of tests better aligns our sample with the natural target for our approach.

Our benchmark thus consists of a total of 200 test classes combined from the development and evaluation sets. The development set contains 442 unit tests and the evaluation set contains 1398 unit tests. The larger number of tests in our evaluation set is the consequence of our constraint to only select classes with at least ten test methods.

Table 6 Benchmark Composition. Each row represents one convention family for which Sif44J provides a fragment extraction rule. For both the development and evaluation sets, the columns labeled *Tests* report the number of tests where the corresponding convention is expected, and the columns labeled *Classes* report in how many different classes these tests are located. The rows are presented in decreasing number of occurrences in the evaluation set.

Convention	**Development Set**		**Evaluation Set**	
	Tests	**Classes**	**Tests**	**Classes**
Method Only	95	31	367	54
Method–State	50	14	251	49
Result–State	59	12	187	26
Method–State–Result	9	3	87	10
Result Only	29	10	73	16
Abbreviated Method Only	13	10	62	18
Method–Result	8	3	47	19
Class–State	12	4	46	5
State–Scenario–Result	9	2	46	4
State Only	25	9	44	14
AbbreviatedMethod–State	28	6	27	10
Scenario–State	8	5	14	5
Class Only	20	15	8	2
State–Result	10	2	5	3
Scenario–State–Result	4	2	4	1
Result–Method–State	12	1	0	0
Applicable Conventions	391		1268	
+ Inapplicable conventions	51		130	
Total	442		1398	

Table 6 shows the composition of our benchmark in terms of expected convention families. For each convention family for which Sif44J provides a fragment extraction rule, we report the number of corresponding tests and the number of classes in which these tests are distributed. As we are applying Sift4J to randomly-selected test code, some unit tests in the benchmark do not follow any of the conventions we can detect. To capture this important factor of the evaluation, we define *applicable tests* as the set of benchmark tests whose *expected convention* is implemented by the predefined rules. The development set contains 391 applicable tests out of the 442 tests (88.5%), and the evaluation set contains 1268 applicable tests out of the 1398 tests (90.7%).

Our benchmark thus collects a curated collection of tests from existing test suites that exhibits a diversity of test naming practices, with a number of tests for each convention in approximate proportion to the frequency we observed them in our formative study (see Section 3).[7] Given our purposive sampling procedure, we do not claim that the relative proportions in the test naming practices represented in the benchmark are representative of the general population of Java test suites. The complete benchmark is available in our online dataset [31].

5.2 Evaluation Metrics

A data point in our evaluation is the application of Sift4J to a given unit test. The *expected convention (family)* for a unit test is the convention (family) used for the unit test as annotated by the first author (see Table 2). In this section, we henceforth refer to convention families simply as *conventions* for short. The *detected convention* is the convention output by Sift4J.

We answer the research questions in terms of two metrics: accuracy and Cohen's kappa (κ). Accuracy provides a simple overview of the performance of the approach through the ratio of tests for which Sift4J can detect the expected convention. We use two formulations of accuracy. Accuracy_g (*global*) is the ratio of tests for which the detected convention is the expected convention over *all* tests. In contrast, Accuracy_a (*applicable*) is the ratio of tests for which the detected convention is the expected convention over *applicable* tests. The two metrics allow us to evaluate two different aspects of the approach: Accuracy_a represents the performance of the current implementation of Sif4J's predefined rules, while Accuracy_g gives an overview of the performance of the approach we could expect if we deployed it in practice.

In addition to overall performance, we also study the performance for each predefined convention. For this purpose we use Cohen's κ (kappa) statistic [14]. For each convention, we construct a 2×2 confusion matrix that distinguishes *expected* vs. *not-expected* in one dimension and *detected* vs. *not-detected* in the other. We use the κ statistics for this evaluation to mitigate the effect of class imbalance.[8]

[7] As a result of the random sampling, the Result–Method–State convention has no corresponding test in the evaluation benchmark. We accept this limitation as a trade-off for using random sampling to avoid potential bias in the construction of the benchmark, because the rule is a permutation of the Method–State–Result, which is well covered, and because the implementation of the rule was covered in the development set.

[8] For each convention except the most popular ones, most tests will naturally be classified as *not expected*, leading to a class imbalance. In such cases, a large proportion of matches is

Our second research question only considers cases where Sift4J detected the correct convention for a unit test. For such cases, we compute the fragment-level accuracy Accuracy_f as the number of correctly identified fragments over the total number of expected fragments for all tests for which the correct convention was detected in a class.

5.3 Results

We separately present the evaluation results for the development set and evaluation set.

5.3.1 Development Set

The accuracy over *applicable tests* (accuracy_a) is 96.7%, while the accuracy over all tests (accuracy_g) is 85.5%. Table 7 documents the causes of classification errors for the development set. For a given test, a *false negative* corresponds to Sift4J not triggering any rule when one is expected; a *false positive* corresponds to Sift4J triggering a rule when none is applicable, and a *misclassification* corresponds to selecting the incorrect rule (in effect a matching false positive–false negative pair). The table organizes the causes of classification errors in six categories, also discussed in Section 4.4. For each category, we report the total number of occurrences (*Tot.*), which we further break down in terms of the number of occurrences that are *false negatives (FN)*, or misclassifications (*Mis.*). Over 391 applicable tests, we observed 5 false negatives and 8 misclassifications (there were no unmatched false positives).

Table 7 Causes of classification errors in the development set. The columns indicate the total number of occurrences (Tot.), the number of false negatives (FN), the number of misclassificationss (Mis.).

Cause	**Tot.**	**FN**	**Mis.**
Reuse of a term	3	0	3
High level of abstraction	3	2	1
Idiosyncratic naming style	3	1	2
Limitation of the POS Tagger	2	0	2
Thresholding problem	2	2	0
Total	13	5	8

First, we observed that, in three cases, a common term used in a method's name as well as in the name of its declaring class caused a misclassification. Second, we observed in three separate cases that use of high-level language led to ambiguities and corresponding misclassifications. For example, a test named testDiscoveryBlockingDisabled describes the state of the test where a parameter ...discovery.blocking.enabled is set to false. However, this caused a false negative of the *State Only* rule, as the

not informative as they could occur by chance. The κ statistics accounts for this factor so that higher κ values robustly represent higher agreement beyond what can be expected by chance.

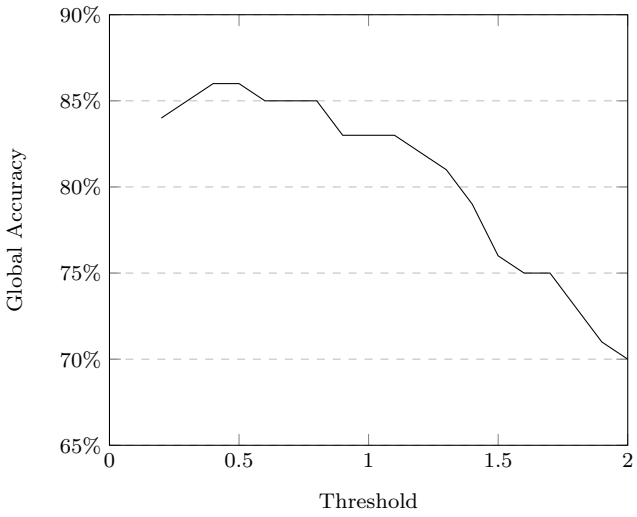

Fig. 6 Sensitivity of threshold to small variations

STATE fragment is described using natural language that inverses the polarity of the state. Third, the use of idiosyncratic names, including uncommon separation tokens in tests and poor production method name, contributed to errors. Fourth, we noted two cases of errors caused by limitations of the POS tagger. One example was a test named isTypeOf_declaredType. In Java programming, declaredType usually refers to the type of a variable used in the declaration, which is expected to be tagged as noun phrase, but the Stanford POS Tagger identified it as a verb phrase. Finally, two errors could be traced to the threshold used for evaluating the similarity between the identified text from the test and the test name impacted the results. For example, a test named testFitForSameInputDifferentQuery was associated with the focal method named fitProcess. In this case, the calculated similarity score between two texts was below the selected threshold, resulting in a false negative.

To determine the threshold value, we conducted a sensitivity analysis by running Sift4J on the development set with different values of threshold and computed the overall accuracy. Figure 6 shows the sensitivity of the threshold to small variation (0.1) on our development set. While we consider the current threshold (0.5) to be a reasonable choice for our data set, the ideal threshold value may vary between projects [32].

In summary, the majority of the classification errors are consistent with the limitations discussed in Section 4.4, and thus confirm opportunities to improve the performance of the tool. For instance, using a POS Tagger designed for software engineering contexts, implementing more convention rules, etc.

Table 8 shows the evaluation results for each convention. We observed κ values greater than 0.8 for each convention rule, indicating that each convention rule is fit for purpose to detect the expected convention [14].

In addition, 604 out of 614 expected information fragments within the development set were correctly identified. The accuracy over fragment-level (Accuracy_f) is thus near perfect (0.98). The few classifications errors we observed were caused by the order of common convention patterns. One example is a test named shouldDo-

DefaultFormatForNestedCaseEndConditionWithFunctionsKeywords, which matches two predefined convention patterns: testShould(\w+)For(\w+) and testShould(\w+)With(\w+). The extraction result is affected by execution order of these patterns. Overall, the results of the evaluation on the development set confirm that the implementation of the predefined rules adequately captures the salient features of the development set.

Table 8 Cohen's Kappa per Convention on the development set. The columns indicate the number of true positives (TP), the number of false positives (FP), the number of true negatives (TN), the number of false negatives (FN).

Convention	TP	FP	TN	FN	κ
Class Only	20	0	422	0	1.00
Result–Method–State	12	0	430	0	1.00
State–Scenario–Result	9	0	433	0	1.00
Method–State–Result	9	0	433	0	1.00
Scenario–State-Result	4	0	438	0	1.00
Method Only	93	0	347	2	0.99
Abbreviated Method–State	27	0	414	1	0.98
Result–State	58	2	381	1	0.97
Result Only	27	0	413	2	0.96
Method–State	49	4	388	1	0.95
State–Result	9	0	432	1	0.95
State Only	24	2	415	1	0.94
Abbreviated Method Only	12	1	428	1	0.92
Method–Result	8	2	432	0	0.89
Scenario–State	8	2	432	0	0.89
Class–State	9	0	430	3	0.85

5.3.2 Evaluation Set

The accuracy over *applicable tests* (accuracy_a) is 94.2% (compared to 96.7% for the development set), while the overall accuracy for all tests (accuracy_g) is 85.4% (compared with 85.5% for the development set). For our evaluation set, we analyzed additional dimensions of the assessment to gain insight on how the approach performs in different contexts.

Applicability by Repository The applicability of Sift4J can be expected to vary from project to project as a consequence of project's adoption of test conventions. Figure 7 summarizes how applicable Sift4J is to the sampled class or classes in each repository in our data set. Each bar represents one of the 76 repositories in our data set. The height of each bar represents the number of applicable tests divided by the number of tests sampled from this repository. For close to half of the repositories (36/76), *all* the tests we sampled have a corresponding rule to detect information fragments. For most of the remaining (37/76), *a majority* of the tests have corresponding rules. For only three of the repositories, a small proportion of the sampled tests have an applicable rule. For the test classes in the project with lowest applicability, google/gson, 11 of the 13 tests have a name that follows the convention family Scenario Only, which we cannot support (see Section 4.1).

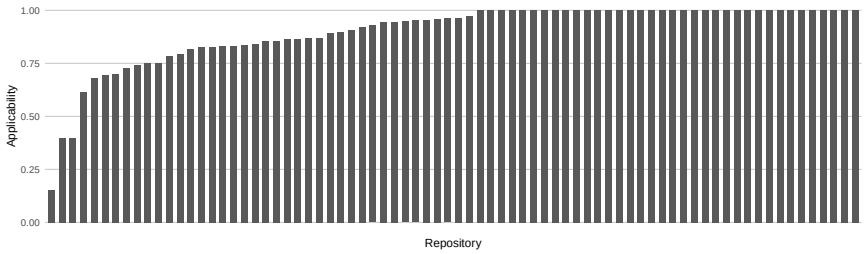

Fig. 7 Sift4J Applicability by Repository. Each bar represents the relative number of applicable tests for a repository.

Performance by Repository We considered the performance of Sift4J in terms of Accuracy$_a$ per repository. Figure 8 summarizes the ratio of applicable tests with a correct classification for the sampled class or classes in each repository. Interestingly, although the Accuracy$_a$ factors out inapplicable tests, the shape of the distribution is similar to the one we observed for applicability.[9] In this case, Sift4J performed with 100% accuracy for the sample tests in 45 of the 76 repositories, with an additional 13 repositories with accuracy above 90%. Only a few repositories exhibit accuracy below 75%. For the repository with poorest performance, mybatis/mybatis-3, the tests follow conventions in the Method Only, Method—State, or similar families, but prefixed the test with the keyword should instead of the more usual test. Customizing the rules to account for this case requires a trivial adjustment.

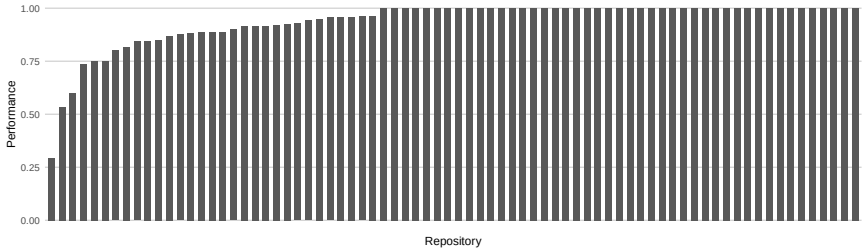

Fig. 8 Sift4J Accuracy$_a$ by Repository. Each bar represents the Accuracy$_a$ per repository.

Extraction Techniques Employed We profiled the execution of Sift4J to report which extraction technique were employed in the identification of each fragment extraction rule. Table 9 provides this data, which confirms that all extraction techniques play a role in the identification of fragments from the names of unit tests in our benchmark. Furthermore, all extraction techniques are associated with a comparable number of incorrect classification, in the range 3.0-6.0%. These numbers

[9] The bars in the figure are ordered by increasing value and the order of repositories differs between the two figures. The Pearson correlation between applicability and Accuracy$_a$ is 0.123.

suggest that, while incremental improvements are possible, the fallback system of heuristics we designed is balanced and fit for purpose.

Table 9 Number of time each fragment extraction technique was used in the applicable rule triggered, for correct vs. incorrect classifications.

Extraction Technique	Correct	Incorrect	Total
Convention Patterns	363	12 (3.5%)	375
Static Analysis	779	24 (3.0%)	803
Grammatical Patterns	172	11 (6.0%)	183
Keywords	233	10 (4.1%)	243

Causes of Classification Errors We observed low applicable accuracy for two specific test classes: NetUtilsTest (36%) and ResourcesTest (29%). The NetUtilsTest class has seven classification failures due to the same misspelling of test as tet. In the ResourcesTest class of the mybatis-3 repository mentioned above, all 12 errors are caused by idiosyncratic names that do not follow any supported convention. Both of these cases point to issues with the naming of tests rather than the engineering of our extraction approach. Table 10 shows the reasons for all failure cases and their occurrences in the evaluation set, comprising 35 false negatives and 39 misclassifications among applicable tests. In general, the causes for classification errors align with those observed in the development set. The predominant cause of errors is idiosyncratic names, characterized by four specific issues: improper use of marker tokens, poor production method names, typographical errors, and variations in word forms. For example, a focal method named click is manually traceable through the terms clicks and clicking in the test name, resulting in a misclassification. The reason *high level of abstraction* impacted the accuracy of the State Only and Result Only convention families. Additionally, the reason *selection of threshold values* predominantly affected the Abbreviated Method Only and Abbreviated Method–State convention families. Table 11 shows the evaluation results for each convention rule on the evaluation set. Compared to the performance of each rule in the development set, the majority of the convention rules maintain a high Cohen's kappa value ($\geq$0.8), except for the Abbreviated Method–State rule. The primary reason for the lower agreement in the Abbreviated Method–State rule can be attributed to the typographical errors and the thresholding effect.

Fragment Classification For the second research question, 1999 out of 2005 expected information fragments within the evaluation set were correctly identified. The accuracy over fragment-level (Accuracy_f) remains nearly perfect. All classification errors are due to the use of different word forms. For example, a test named resolvesRelativeUrls associated with a production method name resolve, however, the use of the third person singular form of the verb leads to a false classification. Overall, the results of applying Sift4J on the evaluation set is comparable to those obtained on the development set, showing that Sift4J can effectively detect the corresponding naming convention and extract the correct sequence of information fragments.

Table 10 Causes of classification errors in the evaluation set. The columns indicate the total number of occurrences (Tot.), the number of false negatives (FN), the number of misclassificationss (Mis.).

Cause	**Tot.**	**FN**	**Mis.**
Reuse of a term	6	1	5
High level of abstraction	17	13	4
Idiosyncratic naming style	33	8	25
Limitation in POS Tagger	4	2	2
Selection of threshold value	14	12	2
Total	74	35	39

Table 11 Cohen's Kappa per Convention on the evaluation set. The columns indicate the number of true positives (TP), the number of false positives (FP), the number of true negatives (TN), the number of false negatives (FN).

Convention Rule	**TP**	**FP**	**TN**	**FN**	κ
Class Only	8	0	1390	0	1.00
State–Scenario–Result	46	0	1352	0	1.00
Scenario–State–Result	4	0	1394	0	1.00
Method Only	354	0	1031	13	0.98
Method–State–Result	83	0	1311	4	0.97
Method–State	236	4	1143	15	0.95
Result–State	183	18	1193	4	0.93
Method–Result	42	2	1349	5	0.92
Result Only	67	7	1318	6	0.91
Abbreviated Method Only	52	1	1335	10	0.90
State–Result	4	0	1393	1	0.89
Class–State	44	9	1343	2	0.88
State Only	35	2	1352	9	0.86
Scenario–State	14	6	1378	0	0.82
Abbreviated Method–State	22	8	1363	5	0.77
Result–Method–State	0	0	1398	0	N/A

5.4 Discussion

The results of applying Sift4J to our benchmark surface a number of important insights. First, the high accuracy we observe validates the design and feasibility of the approach. Specifically, for all the 16 predefined conventions we implemented, Sift4J correctly identifies the tests with the corresponding convention (RQ1) and, given a correct convention, almost all information fragments can be properly retrieved (RQ2). These results are based on applying the approach to 1840 different tests in 200 different test classes sampled from 96 different GitHub repositories of Java projects.

Second, the variation in applicability and performance between repositories highlights that the performance of approach is context-dependent. Figures 7 and 8 show that, while the approach can be directly applied to, and exhibits high performance on, the classes sampled from most repositories in our benchmark, it is not universally the case. The main limitation of our benchmark study is that the sample we use is not representative. For this reason, it is not possible to use our

overall results to *predict* the performance of the approach on a given project. However, the extent to which a given software project will be amenable to automatic extraction of information fragments from tests can be better estimated from a study of its test naming practices informed by our detailed statistics and qualitative analysis of misclassifications. Essentially, a project that makes *systematic use* of naming conventions *that can be analyzed precisely* will be a good match for fragment extraction, whereas a project that does not use naming conventions for tests will not.

Our results also provide useful insights into the relative accuracy of various fragment extraction strategies, as realized by convention rules in our approach. Tables 8 and 11 report on how accurately each convention can be identified. The consistency between the development and evaluation sets is noteworthy: the Class Only convention, as well as the three-fragment rules based on convention patterns, show near perfect precision, whereas the more heuristic rules involving the detection of, e.g., State or Result fragments without the benefit of a convention pattern, fare less well. This insight can help estimate the cost of recovering information from existing projects. For example, for projects that use a convention pattern of the type Scenario–State–Result, converting the fragments into annotations should involve a minimal amount of effort for adjusting misclassifications. In contrast, a project with naming conventions that use more approximate information, such as abbreviations of focal method names in test names, or that rely on the inference of state from grammatical patterns, is likely to involve more work resolving misclassifications.

As for forward engineering, two approaches are possible: either to adopt a systematic convention and to extract fragments automatically, or to encode fragments directly as test metadata using annotations. The main benefit of recovering fragments from test names is that it avoids perturbing the test writing process: developers can continue to write tests as before. The tradeoff in this case revolves around succinctness vs. expressivity of the selected naming conventions. For example, adopting Method Only is easy to apply, but results in less information encoded in test names, whereas three-fragment convention patterns are more verbose, but provide information in multiple dimensions. As for encoding information directly as annotations, the benefit is an immediate and precise encoding of test metadata, at the initial cost of writing the annotations manually.

Our analysis of the misclassifications we encountered provides additional guidance for developers wishing to improve the quality of their test suites. For instance, using a linter to check test prefix markers (i.e., `test`) would already have avoided issues in our benchmark tests. Other issues we noted point to the need for explicit conventions for how to handle situations such as expressing polarity for Boolean variables (e.g., using `NotEnabled` instead of `disabled` for a property `enabled`), avoiding encodings based on grammatical patterns, and ensuring that uncommon conventions have a corresponding implementation (e.g., when using tokens other than `test` to mark tests). By surfacing these varied concerns, our results illustrate how naming tests can be a consequential code quality concern that could deserve attention when developing test suites.

Finally, one observation from our formative study, which is reflected in our benchmark, is that most of the tests we sampled use conventions that encode only one or two fragments (see Table 3). One way to improve the descriptiveness of test names is thus to increase the adoption of naming convention patterns that encode

additional information, such as Method–State–Result. Another avenue could be the combination of effective naming conventions with automated test tagging [15], template-based documentation [18], or summarization techniques (e.g., [16, 21]).

6 Conclusion

Motivated by the observation that test names often encode latent semantic information and the difficulty of maintaining large test suites, we designed Sift4J, a novel rule-based approach to automatically extract the semantic information fragments encoded in the name of a unit test.

The design of Sift4J was informed by the related work on test-to-code traceability and by a formative study of popular Java projects, in which we identified 36 families of test naming conventions that encoded five different types of information: the name of the focal method (possibly abbreviated), the name of the focal class, the input (state) for the test, the expected result, or a scenario describing the behavior under test.

We designed Sift4J as a rule engine to detect 16 of the most popular naming convention families we encountered to automatically extract information fragments from test names, and to convert them into Java annotations. Specifying test metadata as annotation allows tools to analyze and reorganize tests suites based on different *dimensions*, such as the focal method or a certain type of input state (e.g., null values). We illustrated a proof of concept of this potential with a plug-in for an integrated development environment.

We evaluated Sift4J on a benchmark consisting of 1840 different tests in 200 different Java test classes sampled from 96 different GitHub repositories. The results show an accuracy of 94.2% for detecting an applicable convention family when evaluated on the subset of tests not used to develop the approach. A detailed analysis of the classification errors showed that many are due to imprecise terms in test names. We conclude that, for projects that follow good test naming conventions, extracting information fragments from test names is a realistic and effective avenue for enriching test suites with additional metadata.

Acknowledgements The authors are grateful to Jin Guo and the anonymous reviewers for feedback on this work.

Declarations

Funding This work was funded by the Natural Sciences and Engineering Research Council of Canada (NSERC).

Conflicts of Interest The authors have no conflicts of interests to declare that are relevant to the content of this article.

Author Contributions Both authors made substantial contributions to the work.

Data Availability Statement This work is complemented by an online dataset that includes the empirical data collected as part of the research [31].

Ethical Approval and Informed Consent Not applicable.

References

1. Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, page 38–49, 2015. https://doi.org/10.1145/2786805.2786849.
2. Venera Arnaoudova, Laleh M. Eshkevari, Massimiliano Di Penta, Rocco Oliveto, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. Repent: Analyzing the nature of identifier renamings. *IEEE Transactions on Software Engineering*, 40(5):502–532, 2014. https://doi.org/10.1109/TSE.2014.2312942.
3. Sebastian Baltes and Paul Ralph. Sampling in software engineering research: a critical review and guidelines. *Empirical Software Engineering*, 27(94), 2022. https://doi.org/10.1007/s10664-021-10072-8.
4. Dave Binkley, Matthew Hearn, and Dawn Lawrie. Improving identifier informativeness using part of speech information. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, page 203–206, 2011. https://doi.org/10.1145/1985441.1985471.
5. Ermira Daka, José Miguel Rojas, and Gordon Fraser. Generating unit tests with descriptive names or: would you name your children thing1 and thing2? In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 57–67, 2017. https://doi.org/10.1145/3092703.3092727.
6. Mohammad Ghafari, Carlo Ghezzi, and Konstantin Rubinov. Automatically identifying focal methods under test in unit test cases. In *Proceedings of the IEEE 15th International Working Conference on Source Code Analysis and Manipulation*, pages 61–70, 2015. https://doi.org/10.1109/SCAM.2015.7335402.
7. Giovanni Grano, Cristian De Iaco, Fabio Palomba, and Harald C. Gall. Pizza versus pinsa: On the perception and measurability of unit test code quality. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, pages 336–347, 2020. https://doi.org/10.1109/ICSME46990.2020.00040.
8. Michaela Greiler, Arie van Deursen, and Margaret-Anne Storey. Automated detection of test fixture strategies and smells. In *Proceedings of the IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 322–331, 2013. https://doi.org/10.1109/ICST.2013.45.
9. Samir Gupta, Sana Malik, Lori Pollock, and K. Vijay-Shanker. Part-of-speech tagging of program identifiers for improved text-based software engineering tools. In *Proceedings of the 21st International Conference on Program Comprehension*, pages 3–12, 2013. https://doi.org/10.1109/ICPC.2013.6613828.
10. Reid Jon. Unit test naming: The 3 most important parts. Personal blog, Apr 2020. Verified 2024-05-20. https://qualitycoding.org/unit-test-naming/.
11. Petri Kainulainen. Writing clean tests: Naming matters. Personal blog, Jan 2018. Verified 2023-04-24. https://www.petrikainulainen.net/programming/testing/writing-clean-tests-naming-matters/.
12. Pavneet Singh Kochhar, Xin Xia, and David Lo. Practitioners' views on good software testing practices. In *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice*, pages 61–70, 2019. https://doi.org/10.1109/ICSE-SEIP.2019.00015.
13. Ajitesh Kumar. 7 popular strategies: Unit test naming conventions. DZone article, Jun 2021. Verified 2023-04-24. https://dzone.com/articles/7-popular-unit-test-naming.
14. J. Richard Landis and Gary G. Koch. The measurement of observer agreement for categorical data. *Biometrics*, 33(1):159–174, 1977.
15. Boyang Li, Christopher Vendome, Mario Linares-Vasquez, and Denys Poshyvanyk. Aiding comprehension of unit test cases and test suites with stereotype-based tagging. In *Proceedings of the IEEE/ACM 26th International Conference on Program Comprehension*, pages 52–5211, 2018. https://doi.org/10.1145/3196321.3196339.
16. Boyang Li, Christopher Vendome, Mario Linares-Vásquez, Denys Poshyvanyk, and Nicholas A. Kraft. Automatically documenting unit test cases. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 341–352, 2016. https://doi.org/10.1109/ICST.2016.30.

17. Christopher Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven Bethard, and David McClosky. The Stanford CoreNLP natural language processing toolkit. In Kalina Bontcheva and Jingbo Zhu, editors, *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 55–60, 2014. https://doi.org/10.3115/v1/P14-5010.
18. Mathieu Nassif, Alexa Hernandez, Ashvitha Sridharan, and Martin P. Robillard. Generating unit tests for documentation. *IEEE Transactions on Software Engineering*, 48(9):3268–3279, 2022. https://doi.org/10.1109/TSE.2021.3087087.
19. Wyatt Olney, Emily Hill, Chris Thurber, and Bezalem Lemma. Part of speech tagging java method names. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, pages 483–487, 2016. https://doi.org/10.1109/ICSME.2016.80.
20. Roy Osherove. Naming standards for unit tests. Personal blog, Apr 2005. Verified 2023-08-04. https://osherove.com/blog/2005/4/3/naming-standards-for-unit-tests.html.
21. Sebastiano Panichella, Annibale Panichella, Moritz Beller, Andy Zaidman, and Harald C. Gall. The impact of test case summaries on bug fixing performance: An empirical investigation. In *Proceedings of the 38th IEEE/ACM International Conference on Software Engineering*, pages 547–558, 2016. https://dl.acm.org/doi/10.1145/2884781.2884847.
22. Reza Meimandi Parizi. On the gamification of human-centric traceability tasks in software testing and coding. In *Proceedings of the IEEE 14th International Conference on Software Engineering Research, Management and Applications*, pages 193–200, 2016. https://doi.org/10.1109/SERA.2016.7516146.
23. Anthony Peruma, Emily Hu, Jiajun Chen, Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Christian D. Newman. Using grammar patterns to interpret test method name evolution. In *Proceedings of the IEEE/ACM 29th International Conference on Program Comprehension*, page 335–346, 2021. https://doi.org/10.1109/ICPC52881.2021.00039.
24. Abdallah Qusef, Gabriele Bavota, Rocco Oliveto, Andrea De Lucia, and Dave Binkley. Recovering test-to-code traceability using slicing and textual analysis. *Journal of Systems and Software*, 88:147–168, 2014. https://doi.org/10.1016/j.jss.2013.10.019.
25. Abdallah Qusef, Gabriele Bavota, Rocco Oliveto, Andrea De Lucia, and David Binkley. SCOTCH: Test-to-code traceability using slicing and conceptual coupling. In *Proceedings of the IEEE 27th International Conference on Software Maintenance*, page 63–72, 2011. https://doi.org/10.1109/ICSM.2011.6080773.
26. Abdallah Qusef, Rocco Oliveto, and Andrea De Lucia. Recovering traceability links between unit tests and classes under test: An improved method. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 1–10, 2010. https://doi.org/10.1109/ICSM.2010.5609581.
27. Martin P. Robillard, Mathieu Nassif, and Muhammad Sohail. Understanding test convention consistency as a dimension of test quality. *ACM Transactions on Software Engineering and Methodology*, 31(4):39 pages, 2024. https://dl.acm.org/doi/10.1145/3672448.
28. Bart Van Rompaey and Serge Demeyer. Establishing traceability links between unit test cases and units under test. In *Proceedings of the 13th European Conference on Software Maintenance and Reengineering*, pages 209–218, 2009. https://doi.org/10.1109/CSMR.2009.39.
29. Andrew Trenk. Testing on the toilet: Writing descriptive test names. Google Testing Blog, Oct 2014. Verified 2024-05-10. https://testing.googleblog.com/2014/10/testing-on-toilet-writing-descriptive.html.
30. Sinan Wang, Ming Wen, Yepang Liu, Ying Wang, and Rongxin Wu. Understanding and facilitating the co-evolution of production and test code. In *Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering*, pages 272–283, 2021. https://doi.org/10.1109/SANER50967.2021.00033.
31. Ziming Wang and Martin P. Robillard. Data from: Supporting multi-dimensional unit test classification. Zenodo, April 2025. https://doi.org/10.5281/zenodo.15173864.
32. Robert White, Jens Krinke, and Raymond Tan. Establishing multilevel test-to-code traceability links. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 861–872, 2020. https://doi.org/10.1145/3377811.3380921.
33. Dietmar Winkler, Pirmin Urbanke, and Rudolf Ramler. Investigating the readability of test code. *Empirical Software Engineering*, 29(53), 2024. https://doi.org/10.1007/s10664-023-10390-z.
34. Jianwei Wu and James Clause. A pattern-based approach to detect and improve non-descriptive test names. *Journal of Systems and Software*, 168:110639, 2020. https://doi.org/10.1016/j.jss.2020.110639.

35. Jianwei Wu and James Clause. Automated identification of uniqueness in JUnit tests. *ACM Transactions on Software Engineering and Methodology*, 32(1), 2023. https://doi.org/10.1145/3533313.
36. Jianwei Wu and James Clause. A uniqueness-based approach to provide descriptive junit test names. *Journal of Systems and Software*, 205:111821, 2023. https://doi.org/10.1016/j.jss.2023.111821.
37. Benwen Zhang, Emily Hill, and James Clause. Automatically generating test templates from test names (n). In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, pages 506–511, 2015. https://doi.org/10.1109/ASE.2015.68.
38. Benwen Zhang, Emily Hill, and James Clause. Towards automatically generating descriptive names for unit tests. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 625–636, 2016. https://doi.org/10.1145/2970276.2970342.