Multi-dimensional Unit Test Classification

Ziming Wang, School of Computer Science

McGill University, Montreal

June, 2024

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of

Master of Computer Science

Abstract

In software development projects, unit test names contribute to the overall quality of the tests. Developers often encode rich contextual information in the test names to enhance the test readability and maintainability. 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 still be hard to read and maintain, even with descriptive names. To address these limitations, we propose to identify common types of information encoded in test names and prevalent test naming conventions, and 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 extract from test names can include the name of the method under test, a description of the state of the object under test, or the expected result of executing the unit under test. We then demonstrate how to perform multi-dimensional classification of unit tests using this information. Finally, 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.

Abrégé

Dans les projets de développement de logiciels, les noms des tests unitaires contribuent à la qualité globale des tests. Les développeurs encodent souvent des informations contextuelles riches dans les noms des tests pour améliorer la lisibilité et la maintenabilité des tests. Cependant, ces informations manquent de structure formelle et ne peuvent donc pas être systématiquement utilisées pour soutenir les pratiques de développement logiciel telles que la documentation et le refactoring des tests. De plus, les grandes suites de tests peuvent toujours être difficiles à lire et à maintenir, même avec des noms descriptifs. Pour répondre à ces limitations, nous proposons d'identifier les types communs d'informations encodées dans les noms des tests et les conventions de nommage des tests répandues, et d'introduire une nouvelle approche basée sur des règles, appelée Sift4J, pour extraire automatiquement les informations sémantiques latentes encodées dans le nom d'un test unitaire. Les fragments d'information que nous extrayons des noms de tests peuvent inclure le nom de la méthode testée, une description de l'état de l'objet testé, ou le résultat attendu de l'exécution de l'unité testée. Nous démontrons ensuite comment effectuer une classification multidimensionnelle des tests unitaires en utilisant ces informations. Enfin, nous évaluons les performances de Sift4J sur deux échantillons de tests unitaires : notre ensemble de développement et un référentiel d'évaluation précédemment non vu. Les résultats montrent que nous pouvons extraire suffisamment d'informations des noms des tests pour aider à réorganiser de manière significative les tests dans les classes de tests.

Acknowledgements

First and foremost, I express my heartfelt gratitude to my supervisor, Prof. Martin P. Robillard. His invaluable guidance carried me through all stages of this project, from the initial proposal to the final thesis. It is a great honour and privilege to have had the opportunity to work under his supervision. His conscientious academic spirit, high ethical personality and great sense of humour inspires me both in research work and daily life. I believe that I will continue to benefit from this experience throughout my entire life.

I am also deeply thankful to Prof. Robillard for providing me with funding through the Natural Sciences and Engineering Research Council of Canada (NSERC). With this financial support, I am able to fully dedicate myself to my research.

Next, I want to extend thanks to the members of the Software Technology Lab. I am grateful for all the professional knowledge and insightful advice you have shared during our weekly meetings. In addition, you have been my closest friends in Montreal, making my graduate experience memorable.

Finally, and most importantly, I want to thank my parents. Without your endless support and love, it would never have been possible to make this achievement happen. I am especially grateful to my fiancée, Jiayu. You've been there for me when I needed you most, and I could not have done it without you.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Readability and maintainability are the key quality attributes for unit tests [7]. Test method names often have an impact on test suite readability and maintainability [4], 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, elicit the missing tests, etc. Thus, developers often encode rich semantic information in the test names, (e.g., the name of the unit under test, the feature under the test, and the expected outcome of the test). However, the encoding of information along these different *dimensions* is unstructured and unsystematic, and thus prone to inconsistencies and difficult to use by tools. In addition, long test suites with descriptive names can still be hard to read and maintain. To overcome these problems, we investigate three research questions:

RQ 1: What important information do developers commonly include in a test method name?

RQ 2: How can this information be automatically identified?

RQ 3: How can this information help organize a test suite?

In this thesis, we first identify common types of information encoded in test names and 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. Sift4J comprises a collection of semantic fragment extraction rules, each of which is associated with a naming convention. Sift4J uses an ensemble of information extraction techniques that include textual analysis using regular expression, static analysis of the test code, and natural language processing of the test names to convert the information in test names to Java annotation. Furthermore, we developed an IntelliJ plug-in to allow users to browse and organize the tests in a test class according to various dimensions determined by the various information fragments detected.

Finally, we evaluated Sift4J by measuring its accuracy on two samples of unit tests: a 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.

1.1 Contributions

Overall, 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 catalogue of semantic information fragments identified from a sample of Java unit tests;

3. A benchmark of unit test names and their applied naming conventions;

4. A prototype tool called Sift4J that automatically extracts the semantic information fragments from Java unit tests that use the JUnit framework, and an IntelliJ 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.

1.2 Thesis Organization

The remainder of this thesis is structured as follows. Chapter 2 discusses relevant past research and presents a precise formulation of the semantic information fragment detection problem. Chapter 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. Chapter 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. Chapter 5 presents how multi-dimensional test classification is achieved. Chapters 6 and 7 present the design of the evaluation study and the results, respectively. Finally, Chapter 8 presents the conclusions and directions for future work.

Chapter 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 could encode 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, 14, 19, 28]). A common 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 widely. 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., [6,21,23–25,27]). 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 [34]. Their proposal relies on the assumption that the test name would follow a "well-defined grammatical structure" that consists of a "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" [35]. 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 [31] devised a pattern-based approach to compare test names and their corresponding bodies. 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 [32, 33] further leveraged this information and proposed a uniqueness-based approach to generate test names. 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 [22]. 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. [13] conducted open-ended interviews to identify 29 hypotheses that describe characteristics of good test cases, and surveyed 261 practitioners to validate these hypotheses. Key findings revealed that most 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 real practice, for example, running a specific set of tests easily at a time. Several common testing frameworks like JUnit provided

a set of annotations to tag test cases. E.g., in JUnit 5, @Nested, and @Tag annotations were used to help with organizing test suites. @Nested is designed to signal that the annotated class is a nested test class. It can be used to group multiple test methods inside multiple nested classes. Next, @Tag is designed to declare a tag for the annotated test classes or test methods, which can be used to filter which tests are executed for a given test plan. However, these techniques require much human effort and comprehensive understanding on the test class from developers. Another related research is from Li et al. [16], who predefined a catalogue of 21 stereotypes, which are mostly JUnit API-based. And then they developed a prototype 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 utilizing 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 *convention* C. C is a class that describes the naming convention applied to the test, encoding 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. The concatenation of T in the tuples represents an occurrence of the convention class C. In practice, the *context* for a test name is the code base that contains the test together with its necessary dependencies.

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

Method–Result: $\{$(isHorizontal, FOCAL METHOD), (False, EXPECTED 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.

Chapter 3

Types of Semantic Information Fragments

In this chapter, 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 tagging semantic information fragments in unit tests based on existing 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

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

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 1263 test methods from 300 classes.[4] Of these methods, 18 had names that clearly captures 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

[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.

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 accidental 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, 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 3.1 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). Additionally, 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. We thus seek a potential to leverage the most common convention families to extract information fragments encoded in the test names.

Table 3.2 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.

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

Table 3.1: 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

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 thus explores how to resolve ambiguous references to this kind of semantic information.

Table 3.2: Types of Semantic Information Fragments Observed in a Sample of Java Unit Tests

Fragment Type	Description	Obs.	Prop.
METHOD	Refer to the method under test [12]. The method should be called within the test.	464	37%
ABBREVIATED METHOD	Refer to a subset of the tokens that form the name of the focal method. Indicates a test that may be broader in scope that the focal method itself.	96	8%
CLASS	Refer to the class under test. [30]	82	7%
STATE	Refers to input state related to FOCAL METHOD	630	51%
RESULT	Refers to the expected outcome of the test case, other than EXCEPTION [12].	428	34%
SCENARIO	A general description of the focus of the test when no category applies that would be more specific. [12]	225	18%

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 a dual-coding approach with inter-rater reliability calculations 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.

Chapter 4

Extracting Semantic Information From Tests

In this chapter, we contribute the design and implementation 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 3.1. The input to Sift4J is a test file and associated code base. The output is an updated version of the input test file with annotations indicating any detected information fragment. The listing of Figure 4.1 shows an example of unit test annotated with Sift4J.

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 4.2 provides a simplified view of the essential elements of the Sift4J design.

```
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  }
```

Figure 4.1: A Sample Unit Test with Annotated Semantic Information Fragments

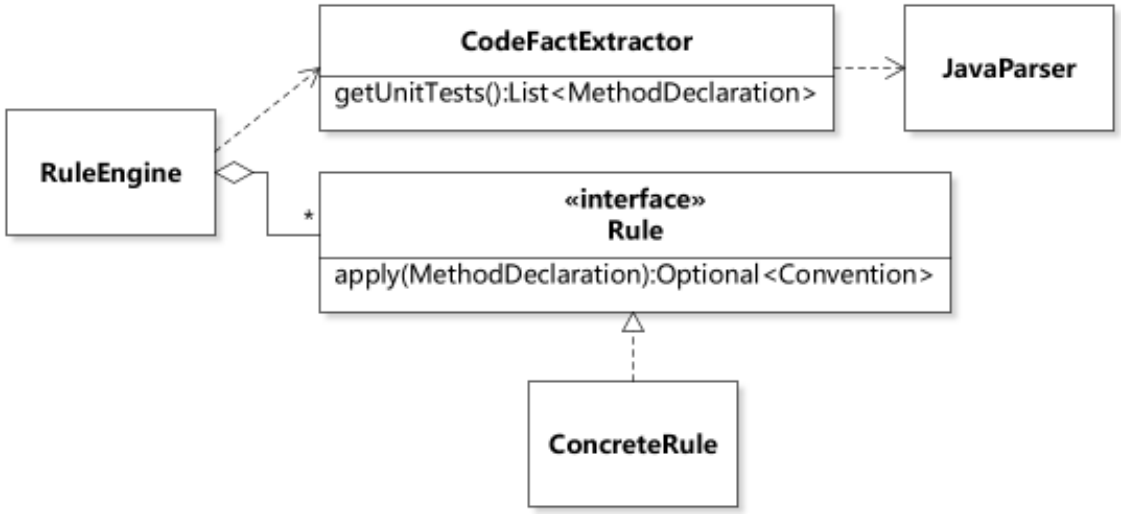

Figure 4.2: Sift4J Overall Design (Simplified)

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.[1] 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 names 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

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

representing a unit test as input. Applying a rule returns an optional *convention*[2] encoding five potentially-empty string instances representing the types of semantic fragments we identified in our formative study (see Section 2.2). In our design, an interface Rule is extended by two categories:[3] OneFragmentConventionRule and MultipleFragmentsConventionRule. A rule is defined by extending the correspond 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 3.1). However, in the list of 18 target conventions, three were not amenable to automatic detection via heuristics: *Scenario Only*, *Scenario–Result*, and *Result–Method–State*. Because of a lack of structure and constraints for expressing such conventions (and in particular scenarios), there is no explicit feature we can rely on to design extraction rules for these families. We implemented support for extracting information fragments for all 15 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*.

[2]Specifically an instance of a class Convention wrapped in an Optional type that remains empty if the rule is not applicable.

[3]implemented as two different abstract classes in practice

Common Convention Pattern 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 *common convention patterns*. We simply use 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. The resulting sequence of semantic fragments is extracted as {(EmptySets: State), (ExpectNoChanges: 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 depend 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 4.1 provides additional details.

Grammatical Relations 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 is likely to describe the input State of the focal method (e.g., test_isHorizontal_withNull). Table 4.2 documents the grammatical relations 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.

Keywords We also leverage the simple heuristic that certain terms in a method name can indicate the presence of specific types of information fragments [22]. E.g., 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. A second example might be the term is a proper noun, e.g., a test named testDetermineSampleSize_PNG, "PNG" refers to a type of image, when it appears after the method under test, then it is likely to suggest that "PNG" is the input state of the focal method.

Table 4.3 reports the subset of techniques we employ for each rule.

4.2.1 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 checking the least ambiguous cases first (i.e., common convention patterns), and then falling back to other alternatives as necessary. In the case of extraction rules for multiple fragments, we may need to take into account the partial matching of the test's name by one technique when applying other technique. For this reason, the rule metaheuristic differs slightly for rules to extract a single fragment (Algorithm 1) from rules to extract more than one fragment (Algorithm 2).

Algorithm 1 One-Fragment Convention Rule Extraction Algorithm

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

$n \leftarrow$ *Unit Test Name*
if n follows a *Common Convention Pattern* **then**
    $f \leftarrow$ APPLYREGULAREXPRESSION(n)
    return BUILDCONVENTION(f)
end if
$n \leftarrow$ PREPROCESS(n) ▷ Remove underscores and "test" related filler words
$f \leftarrow$ APPLYTESTTOCODETRACEBILITY(U)
if f matches n **then**
    return BUILDCONVENTION(f)
else if n starts with *Special Term* **then**
    return BUILDCONVENTION(f)
end if
return Empty

4.2.2 Limitations of the rules

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 (though the rule family employed to detect it), the use of a rule-base approach

Algorithm 2 Two-Fragments Convention Rule Extraction Algorithm

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

$n \leftarrow$ *Unit Test Name*
if n follows a *Common Convention Pattern* **then**
    $f1, f2 \leftarrow$ APPLYREGULAREXPRESSION(n)
    return BUILDCONVENTION(f1, f2)
end if
$n \leftarrow$ PREPROCESS(n) ▷ Remove underscores and "test" related filler words
$f1, f2 \leftarrow$ APPLYTESTTOCODETRACEABILITY(U)
if n starts with $f1 \wedge$ ends with $f2$ **then**
    return BUILDCONVENTION(f1, f2)
end if
if n starts with $f1 \vee$ ends with $f2$ **then**
    $f \leftarrow$ the matched fragment
    $remain \leftarrow$ remove f from n
    if $remain$ follows *Grammatical Relation* $\vee$ starts with *Special Term* **then**
        return BUILDCONVENTION(f, remain)
    end if
end if
return Empty

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-base systems applied to our context. First, not all information can be encoded by following simple conventions. Second, a heuristic approach to match natural language is ambiguous and incomplete 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, and SCENARIO (see Table 3.1). 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 indicates that the test case is validating the basic functionality for a method, Sift4J will be unable to establish a connection between the test name an

any fragment. This limitation is compounded by the reality that, even when a project uses a well-defined convention, it is possible that not all test names consistently follow the naming convention. Consistency is in particular impacted by the challenges of co-evolving test and code [29]. 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 METHODONLY 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 is of a test named testFloorDoubleNumber, whose focal class and focal method are both named Floor (see 4.3).

The third limitation is that the implementation of all four of our extraction techniques (Section 4.2) impacts the performance of the approach. For *Textual Patterns*, the implementation needs to include patterns used in a project for the approach to perform well. Similarly, the *Keywords* approach will be sensitive to the glossary used as hints that certain tokens represent certain types of fragments. The *static analysis* technique relies on the correct parsing and type resolution of incomplete Java source code, which is itself an approximate process. For example, we rely on the JavaParser built-in JavaSymbolSolver, to resolve overloaded method calls. However, the developers of JavaParser have observed potential bugs[4] in the library, possibly caused by lambda functions or variadic parameters. As for matching the names of detected methods to the test name, we rely on a threshold value. However, we conducted a sensitivity analysis to ensure we were working with a optimal value (see Section 7.1 for details). Finally, a word may have a different part-of-speech (POS) tag than usual in a software-specific context [3, 9, 18], which could negatively impact the result of *Grammatical Relations* technique. However, the performance of the Stanford Part-of-Speech Tagger has previously been considered

[4] https://github.com/javaparser/javaparser/issues/1643#issuecomment-396492324

Figure 4.3: Example of reuse term in information semantic fragment.

```
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 }
```

satisfactory on analysing the grammar pattern of software identifiers [2, 22, 31]. 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 approaches first and only rely on less precise alternatives when no other option succeeds.

Table 4.1: Static Analysis Strategies for Extracting Semantic Fragments

Fragment Type	Extraction Strategy	Tagged Text
METHOD	Combine a set of independent heuristics to produce a score following the strategy of White et al. [30]. Obtain the name of all methods called directly within the body of the test, compute four case-insensitive similarity measures between the name of the method called and the name of the unit test, and add the results. The similarity measures are: exact name match, exact name containment, Levenshtein distance, and longest common subsequence.	The name of the called method identified as similar to the test name.
CLASS	Use the same approach 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 as part of the test name.	The name of the argument or the generated state description identified as similar to the test name.
RESULT	Generate a result description based on the assert statement type following the strategy of Zhang. [35]. 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 as part of the test name. In addition, Exception is a special type of Result of unit test, we used three common JUnit framework error handling mechanisms to extract the exception thrown. Specifically, 1. Use the expected attribute of JUnit's @Test annotation 2. Use try-catch idiom with a call to JUnit's fail method in the catch block 3. Use JUnit's assertThrows method	The generated result description or the name of the exception identified as similar to the test name

Table 4.2: Grammatical Relations between Semantic Information Fragments

Rule	Pattern	Example
METHOD–STATE	Method + NP Method + PP Method + ADJP	edgesConnecting_disconnectedNodes decrementByNegativeDelta testGetInReplyTo_empty
ABBREVIATED METHOD–STATE	Abbreviated Method + NP Abbreviated Method + PP Abbreviated Method + ADJP	As above
RESULT–STATE	Result + NP Result + PP Result + ADJP	As above
METHOD–RESULT	Method + VP	isValid_shouldValidateConfigRepo
STATE–RESULT	State + VP	aUUIDStringReturnsAUUIDObject
SCENARIO–STATE	NP + State	cycleOfMixedWithImmutableRoot
RESULT–STATE	VP + State	testReturnsFalseIfFinishingFails

Table 4.3: Extraction Techniques Applied in Predefined Rule Set

Convention Type	Common Convention Pattern	Test-to-Code Traceability	Grammatical Relation	Special Term
METHOD ONLY	No	Yes	No	No
STATE ONLY	No	Yes	No	Yes
METHOD–STATE	Yes	Yes	Yes	Yes
RESULT ONLY	Yes	Yes	No	Yes
RESULT–STATE	Yes	Yes	Yes	Yes
SCENARIO–STATE	Yes	No	No	No
ABBREVIATED METHOD ONLY	No	Yes	No	No
METHOD–STATE–RESULT	Yes	No	No	No
CLASS ONLY	No	Yes	No	No
ABBREVIATED METHOD–STATE	Yes	Yes	Yes	Yes
STATE–RESULT	Yes	Yes	No	Yes
METHOD–RESULT	No	Yes	Yes	Yes
BAD CONVENTION	No	No	No	Yes
SCENARIO–STATE–RESULT	Yes	No	No	Yes
RESULT–METHOD–STATE	Yes	No	No	No
STATE–SCENARIO–RESULT	Yes	No	No	No

Chapter 5

Multi-dimensional Test Classification

This chapter presents how multi-dimensional classification is achieved leveraging the semantic fragments. It describes the implementation of the plug-in built upon Sift4J and presents a demonstration on a real test file.

5.1 Overview of Multi-dimensional Test Classification

Once unit tests are annotated with semantic information fragments (as illustrated in Figure 4.1), 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).

5.2 Sift4J Plug-in

As a proof of concept, we implemented a sample 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 value observed in the test file (e.g., focal method). The plug-in also supports grouping tests in terms of multiple dimensions (for example, first by focal method, then input state). Although grouping tests by multiple levels 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.

5.3 Example of Using Sift4J Plug-in

We illustrate the workflow supported by the Sift4J plug-in with a walk-through of a relatively simple test file called CollectionUtilsTest.java[1] (see Figure 4.1). This class contains tests of the miscellaneous collection utility methods used in the corresponding project. 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 presented in the bottom console organized in a method-by-fragment table (see Figure 5.1). After running Sift4J on this test class, each test case is correctly tagged with annotations that encode semantic fragments. However, use of the Sift4J plug-in is independent from the performance of the automated fragment extraction process. For imperfect fragment extraction outcomes, developers can adjust the fragment annotations

[1] https://github.com/perwendel/spark/blob/54079b0f95f0076dd3c440e1255a7d449d9489f1/src/test/java/spark/utils/CollectionUtilsTest.java/

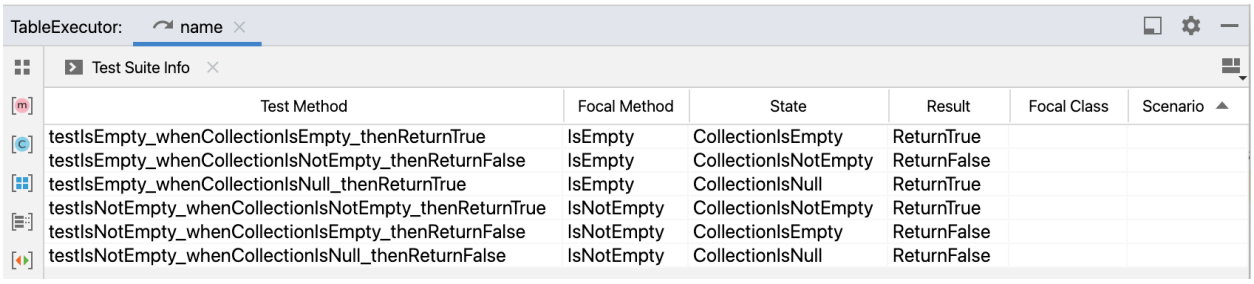

Figure 5.1: The console view of the plug-in running on the sample test suite. Buttons on the left side from top to bottom are: Classify by Default, Classify by Method Fragment, Classify by Class Fragment, Classify by State Fragment, Classify by Result Fragment, Classify by Scenario Fragment.

in 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).

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 correspond button. Each classification strategy prioritizes grouping unit tests based on a different type of semantic fragment. In the case of CollectionUtilsTest, the test cases are organized 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, we thus generate a new nested class to group the test cases accordingly. A similar process is followed in the isNotEmpty class. This grouping process continues until no test cases within the enclosing class share the same fragment value. The complete code of the CollectionUtilsTest.java file as well as different versions produced by the Sift4J plug-in are available in Appendix C, D, E, F, G.

Chapter 6

Evaluation

In this chapter, our goal was to evaluate Sift4J as an initial assessment of 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 [26]. 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?

RQ1: For a correctly identified convention, how effective is Sift4J at extracting semantic information fragments encoded in test names?

6.1 Evaluation Benchmark

In developing the approach we leveraged a *development set* consisting of all the tests in 100 Java test classes. For each test class included in the development set, we had recorded the name of the selected repository and its version number, 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. The development set is documented in Table A.1 in Appendix A.

To evaluate the approach on unseen data, we created an *evaluation set* of 100 Java test classes by randomly selecting 100 additional (unseen) 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 for two reasons. First, we wanted to support an analysis of the performance on a per-class basis, which is only insightful if there is a minimum number of tests in the class. Second, our multi-dimensional test classification approach is only valuable for 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. Table B.1 in Appendix B lists the test classes in our evaluation set.

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.

6.2 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 3.1). In this section, we henceforth refer to convention families simply as *conventions* for short. The *detected convention* is the convention output by Sift4J.

As we are applying Sift4J to unseen, randomly-selected test code, we anticipate that some unit tests will 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%).

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. $\mathrm{Accuracy}_g$ (*global*) is the ratio of tests for which the detected convention is the expected convention over *all* tests. In contrast, $\mathrm{Accuracy}_a$ 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: $\mathrm{Accuracy}_a$ provides sense of the performance of the current implementation of Sif4J's predefined rules, while $\mathrm{Accuracy}_g$ gives an overview of the performance of the approach we could expect if we deployed it in practice. We compute the accuracy metrics both globally (i.e., over applicable/all tests across all test classes), and on a per-class basis (i.e., over applicable/all tests in a given test class).

In addition to overall performance, we also study the performance for each predefined convention. For this purpose we use Cohen's κ (kappa) statistic [15]. 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.[1]

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 $\mathrm{Accuracy}_\mathrm{f}$ 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.

[1]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 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.

Chapter 7

Results and Discussions

In this chapter, We separately present the evaluation results for the *development set* and *evaluation set*.

7.1 Development Set

The accuracy over *applicable tests* (accuracy_a) is 97%, while the accuracy over all tests (accuracy_g) is 86% (see Appendix A). Table 7.1 documents the causes of classification errors for the development set. The table organizes the causes of classification errors in six categories, also discussed in Section 4.2.2. 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)*, *false positives (FP)*, or misclassifications (*Mis.*). 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). Over 391 applicable tests, we observed 5 false negatives and 8 misclassifications.

In these cases, 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,

Table 7.1: Causes of classification errors in the development set

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

in three cases 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 in a false negative of the *StateOnly* rule, as the *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. Next, we noted two cases of errors caused by limitations of the POS tagger. For example, a test named isTypeOf_declaredType. In Java programming, declaredType usually refers to the type of 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. To determine this value, we conducted a sensitivity analysis by running Sift4J on the development set with different values of threshold and computed the overall accuracy. 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. Figure 7.1 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 [30].

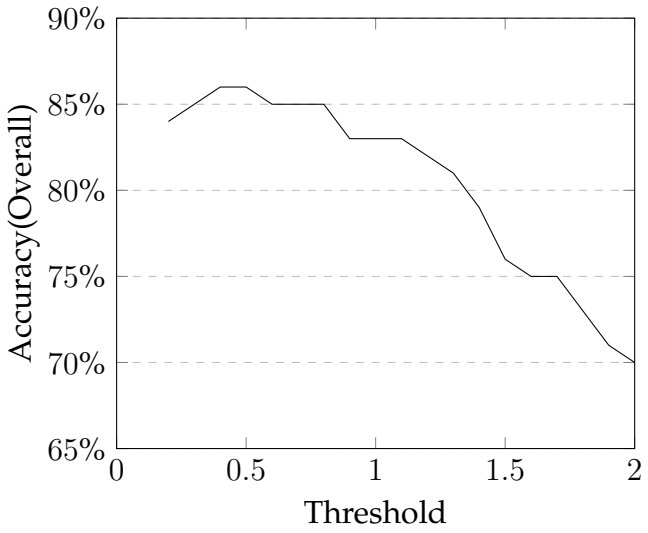

Figure 7.1: Sensitivity of threshold to small variations

In summary, the majority of the classification errors are consistent with the limitations discussed in Section 4.2.2, 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 7.2 shows the evaluation results for each convention. Notably, the κ value for each convention rule is greater than 0.8, indicating that each convention rule works almost perfectly to detect the expected convention [15].

In addition, 604 out of 614 expected information fragments within the development set were correctly identified. The accuracy over fragment-level ($\mathrm{Accuracy_f}$) is thus near perfect (0.98). The few classifications errors we observed were caused by the order of common convention patterns. E.g., a test named shouldDoDefaultFormatForNestedCaseEndConditionWithFunctionsKeywords, 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 show the predefined rules can effectively detect the correspond naming convention and extract the correct sequence of information fragments.

Table 7.2: 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 Rule	TP	FP	TN	FN	κ
Method Only	93	0	347	2	0.99
Result–State	58	2	381	1	0.97
Method–State	49	4	388	1	0.95
Result Only	27	0	413	2	0.96
Abbreviated Method–State	27	0	414	1	0.98
State Only	24	2	415	1	0.94
Class Only	20	0	422	0	1.00
Abbreviated Method Only	12	1	428	1	0.92
Result–Method–State	12	0	430	0	1.00
Class–State	9	0	430	3	0.85
State–Result	9	0	432	1	0.95
State–Scenario–Result	9	0	433	0	1.00
Method–State–Result	9	0	433	0	1.00
Method–Result	8	2	432	0	0.89
Scenario–State	8	2	432	0	0.89
Scenario–State-Result	4	0	438	0	1.00

7.2 Evaluation Set

The accuracy over *applicable tests* (accuracy_a) is 94% (compared to 97% for the development set), while the overall accuracy for all tests (accuracy_g) is 85.4% (compared with 85.5% for the development set, see Appendix B). Notably, we observed relatively low applicable accuracy for two specific test classes: NetUtilsTest (36%) and ResourcesTest (29%). The NetUtilsTest class has seven classification failures due to typographical errors. For example, a test named tetGetIPV6HostAndPort_ReturnHostPort contained a misspelling of test as tet. In the ResourcesTest class, all 12 errors are caused by idiosyncratic names. For example, tests are prefixed with should and followed by the focal method's name.

Table 7.3 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 filler words, 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 in information fragments* notably 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. Despite the larger number of tests in the evaluation set, the alignment of failure reasons with those in the development set underscores that our evaluation effectively highlights the limitations of Sift4J.

Table 7.4 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 is attributed to the typographical errors and the thresholding effect.

For the second evaluation question, 1999 out of 2005 expected information fragments within the evaluation set were correctly identified. The accuracy over fragment-level ($\mathrm{Accuracy_f}$) 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 correspond naming convention and extract the correct sequence of information fragments.

Table 7.3: Causes of classification errors in the evaluation set

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 7.4: 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
Method Only	354	0	1031	13	0.98
Method–State	236	4	1143	15	0.95
Result–State	183	18	1193	4	0.93
Method–State–Result	83	0	1311	4	0.97
Result Only	67	7	1318	6	0.91
Abbreviated Method Only	52	1	1335	10	0.90
Method–Result	42	2	1349	5	0.92
Class–State	44	9	1343	2	0.88
State–Scenario–Result	46	0	1352	0	1.00
State Only	35	2	1352	9	0.86
Abbreviated Method–State	22	8	1363	5	0.77
Scenario–State	14	6	1378	0	0.82
Class Only	8	0	1390	0	1.00
State–Result	4	0	1393	1	0.89
Scenario–State–Result	4	0	1394	0	1.00
Result–Method–State	0	0	1398	0	1.00

Chapter 8

Conclusions

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. Our formative study identified five common types of information and prevalent test naming conventions. We identified six common types of information fragments, including METHOD, ABBREVIATED METHOD, CLASS, STATE, RESULT, and SCENARIO. We also observed eighteen types of prevalent naming conventions in our sample tests, which cover 96% of them. The *Method Only* family had the most observations. With Sift4J, we further contributed a solution to manage large test suites through multi-dimensional classification. Our observation of an accuracy of 94% in naming convention detection and near-optimum accuracy when extracting fragments on an unseen sample of Java tests demonstrates the practical applicability of the approach to legacy code, in addition to being usable in forward-engineering scenarios. Although the current version of the tool focuses on Java tests using JUnit framework, the tool's architecture is language-independent.

8.1 Future Work

Our comprehensive evaluation revealed that Sift4J can extract sufficient information from test names to assist in meaningfully reorganizing the tests in test classes. Two promising directions for future work in this area are test convention consistency and test refactoring. Currently, our approach can detect naming conventions applied to test names, if they are implemented in the predefined rule set. It will be interesting to implement more convention rules and explore how to our approach can be integrated with the current static analysis tools to measure the test naming convention consistency. As for test refactoring, we provided a new refactoring strategy to help developers manage large test suites, which can naturally increase the cohesion between tests by listing them in a meaningful order within a test class, among others. However, future work is needed to evaluate how much this classification strategy can improve the test quality in terms of readability and maintainability, potentially using a set of software metrics [8, 20].

Bibliography

[1] ALLAMANIS, M., BARR, E. T., BIRD, C., AND SUTTON, C. Suggesting accurate method and class names. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering* (2015), p. 38–49. https://doi.org/10.1145/2786805.2786849.

[2] ARNAOUDOVA, V., ESHKEVARI, L. M., PENTA, M. D., OLIVETO, R., ANTONIOL, G., AND GUÉHÉNEUC, Y.-G. Repent: Analyzing the nature of identifier renamings. *IEEE Transactions on Software Engineering 40*, 5 (2014), 502–532. https://doi.org/10.1109/TSE.2014.2312942.

[3] BINKLEY, D., HEARN, M., AND LAWRIE, D. Improving identifier informativeness using part of speech information. In *Proceedings of the 8th Working Conference on Mining Software Repositories* (2011), p. 203–206. https://doi.org/10.1145/1985441.1985471.

[4] BUTLER, S., WERMELINGER, M., YU, Y., AND SHARP, H. Exploring the influence of identifier names on code quality: An empirical study. In *Proceedings of the 14th European Conference on Software Maintenance and Reengineering* (2010), pp. 156–165. https://doi.org/10.1109/CSMR.2010.27.

[5] DAKA, E., ROJAS, J. M., AND FRASER, G. 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* (2017), pp. 57–67. https://doi.org/10.1145/3092703.3092727.

[6] GHAFARI, M., GHEZZI, C., AND RUBINOV, K. 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* (2015), pp. 61–70. https://doi.org/10.1109/SCAM.2015.7335402.

[7] GRANO, G., DE IACO, C., PALOMBA, F., AND GALL, H. C. 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* (2020), pp. 336–347. https://doi.org/10.1109/ICSME46990.2020.00040.

[8] GREILER, M., VAN DEURSEN, A., AND STOREY, M.-A. Automated detection of test fixture strategies and smells. In *Proceedings of the IEEE Sixth International Conference on Software Testing, Verification and Validation* (2013), pp. 322–331. https://doi.org/10.1109/ICST.2013.45.

[9] GUPTA, S., MALIK, S., POLLOCK, L., AND VIJAY-SHANKER, K. Part-of-speech tagging of program identifiers for improved text-based software engineering tools. In *Proceedings of the 21st International Conference on Program Comprehension* (2013), pp. 3–12. https://doi.org/10.1109/ICPC.2013.6613828.

[10] JON, R. Unit test naming: The 3 most important parts. Personal blog, Apr 2020. Verified 2024-05-20. https://qualitycoding.org/unit-test-naming/.

[11] KAINULAINEN, P. 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] KHORIKOV, V. *Unit Testing Principles, Practices, and Patterns*. January 2020.

[13] KOCHHAR, P. S., XIA, X., AND LO, D. Practitioners' views on good software testing practices. In *Proceedings of the IEEE/ACM 41st International*

Conference on Software Engineering: Software Engineering in Practice (2019), pp. 61–70. https://doi.org/10.1109/ICSE-SEIP.2019.00015.

[14] KUMAR, A. 7 popular strategies: Unit test naming conventions. DZone article, Jun 2021. Verified 2023-04-24. https://dzone.com/articles/7-popular-unit-test-naming.

[15] LANDIS, J. R., AND KOCH, G. G. The measurement of observer agreement for categorical data. *Biometrics 33*, 1 (1977), 159–174.

[16] LI, B., VENDOME, C., LINARES-VASQUEZ, M., AND POSHYVANYK, D. 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* (2018), pp. 52–5211. https://doi.org/10.1145/3196321.3196339.

[17] MANNING, C., SURDEANU, M., BAUER, J., FINKEL, J., BETHARD, S., AND MCCLOSKY, D. The Stanford CoreNLP natural language processing toolkit. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations* (2014), K. Bontcheva and J. Zhu, Eds., pp. 55–60. https://doi.org/10.3115/v1/P14-5010.

[18] OLNEY, W., HILL, E., THURBER, C., AND LEMMA, B. Part of speech tagging java method names. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution* (2016), pp. 483–487. https://doi.org/10.1109/ICSME.2016.80.

[19] OSHEROVE, R. 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.

[20] PALOMBA, F., PANICHELLA, A., ZAIDMAN, A., OLIVETO, R., AND DE LUCIA, A. Automatic test case generation: what if test code quality matters? In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (2016), p. 130–141. https://doi.org/10.1145/2931037.2931057.

[21] PARIZI, R. M. 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* (2016), pp. 193–200. https://doi.org/10.1109/SERA.2016.7516146.

[22] PERUMA, A., HU, E., CHEN, J., ALOMAR, E. A., MKAOUER, M. W., AND NEWMAN, C. D. Using grammar patterns to interpret test method name evolution. In *Proceedings of the IEEE/ACM 29th International Conference on Program Comprehension* (2021), p. 335–346. https://doi.org/10.1109/ICPC52881.2021.00039.

[23] QUSEF, A., BAVOTA, G., OLIVETO, R., DE LUCIA, A., AND BINKLEY, D. SCOTCH: Test-to-code traceability using slicing and conceptual coupling. In *Proceedings of the IEEE 27th International Conference on Software Maintenance* (2011), p. 63–72. https://doi.org/10.1109/ICSM.2011.6080773.

[24] QUSEF, A., BAVOTA, G., OLIVETO, R., DE LUCIA, A., AND BINKLEY, D. Recovering test-to-code traceability using slicing and textual analysis. *Journal of Systems and Software 88* (2014), 147–168. https://doi.org/10.1016/j.jss.2013.10.019.

[25] QUSEF, A., OLIVETO, R., AND DE LUCIA, A. Recovering traceability links between unit tests and classes under test: An improved method. In *Proceedings of the IEEE International Conference on Software Maintenance* (2010), pp. 1–10. https://doi.org/10.1109/ICSM.2010.5609581.

[26] ROBILLARD, M. P., NASSIF, M., AND SOHAIL, M. Understanding test convention consistency as a dimension of test quality. *ACM Transactions on Software Engineering and Methodology* (2024), Accepted 2024–05–22.

[27] ROMPAEY, B. V., AND DEMEYER, S. Establishing traceability links between unit test cases and units under test. In *Proceedings of the 13th European Conference on Software Maintenance and Reengineering* (2009), pp. 209–218. https://doi.org/10.1109/CSMR.2009.39.

[28] TRENK, A. 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.

[29] WANG, S., WEN, M., LIU, Y., WANG, Y., AND WU, R. Understanding and facilitating the co-evolution of production and test code. In *Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering* (2021), pp. 272–283. https://doi.org/10.1109/SANER50967.2021.00033.

[30] WHITE, R., KRINKE, J., AND TAN, R. Establishing multilevel test-to-code traceability links. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (2020), pp. 861–872. https://doi.org/10.1145/3377811.3380921.

[31] WU, J., AND CLAUSE, J. A pattern-based approach to detect and improve non-descriptive test names. *Journal of Systems and Software 168* (2020), 110639. https://doi.org/10.1016/j.jss.2020.110639.

[32] WU, J., AND CLAUSE, J. Automated identification of uniqueness in JUnit tests. *ACM Transactions on Software Engineering and Methodology 32*, 1 (2023). https://doi.org/10.1145/3533313.

[33] WU, J., AND CLAUSE, J. A uniqueness-based approach to provide descriptive junit test names. *Journal of Systems and Software 205* (2023), 111821. https://doi.org/10.1016/j.jss.2023.111821.

[34] ZHANG, B., HILL, E., AND CLAUSE, J. Automatically generating test templates from test names (n). In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering* (2015), pp. 506–511. https://doi.org/10.1109/ASE.2015.68.

[35] ZHANG, B., HILL, E., AND CLAUSE, J. Towards automatically generating descriptive names for unit tests. In *Proceedings of the 31st IEEE/ACM*

International Conference on Automated Software Engineering (2016), pp. 625–636. https://doi.org/10.1145/2970276.2970342.

Appendix A

Evaluation Results on Development Set

Table A.1: Accuracy per Test Class on Development Set

Repository	**Commit**	**Test Class**	accuracy_a	accuracy_g	accuracy_f
alibaba/easyexcel	2d5db89	PoiFormatTest	1.00	1.00	1.00
TheAlgorithms/Java	e9bbf35	FloorTest	0.00	0.00	1.00
bumptech/glide	0e6912a	UnitTransformationTest	1.00	0.50	1.00
google/auto	26f073f	MoreTypesIsTypeOfTest	0.80	0.80	1.00
facebook/fresco	91a0935	CloseableBitmapTest	1.00	0.50	1.00
alibaba/nacos	de5ec35	AbstractNamingClientProxyTest	1.00	1.00	1.00
apache/zookeeper	4b1b33e	RequestThrottlerTest	1.00	0.75	1.00
apache/kafka	6010513	KTableKTableForeignKeyJoinMaterializatior	1.00	1.00	1.00
alibaba/nacos	8292d9e	DistroConfigTest	1.00	0.83	1.00
alibaba/fastjson	e8ec59e	TestIssue3671	1.00	1.00	1.00
zaproxy/zaproxy	81ee5d0	PostBasedAuthenticationMethod-TypeUnitTest	1.00	1.00	1.00
netty/netty	f43a9dd	OsClassifiersTest	1.00	1.00	1.00
alibaba/Sentinel	3832f2b	CommandCenterTest	NA	0.00	1.00
apache/dubbo	372f6a6	CommandHelperTest	1.00	1.00	1.00
apache/dubbo	0e66de1	SpringContainerTest	1.00	1.00	1.00
eclipse-vertx/vert.x	62c3996	URLBundleFileResolverTest	1.00	1.00	1.00

apache/kafka	ababc42	SessionWindowedKStreamImplTest	1.00	1.00	1.00
facebook/fresco	b7778fc	AshmemMemoryChunkPoolTest	1.00	1.00	1.00
antlr/antlr4	e1455c0	TestInterpreterDataReader	1.00	1.00	1.00
ben-manes/caffeine	a7d31f5	TypesafeConfigurationTest	1.00	1.00	1.00
apache/cassandra	4526b3f	TopPartitionsTest	NA	0.00	1.00
apache/flink	80cd04a	PythonTableFunctionOperatorTestBase	1.00	0.60	1.00
alibaba/fastjson	7abc84f	RectangleTest	NA	0.00	1.00
apache/flink	94915d6	AggregatedTaskDetailsInfoTest	NA	0.00	1.00
codecentric/spring-boot-admin	568d102	BuildVersionTest	1.00	1.00	1.00
mybatis/mybatis-3	e9a0805	ReflectorTest	1.00	1.00	1.00
dromara/hutool	28fefde	ZipWriterTest	1.00	1.00	1.00
alibaba/canal	63407dc	PhoenixSyncTest	1.00	1.00	1.00
dianping/cat	732b7ee	MetricTest	1.00	1.00	1.00
antlr/antlr4	e1455c0	TestCharSupport	1.00	1.00	1.00
apache/kafka	d3130f2	SaslExtensionsTest	0.67	0.67	1.00
zxing/zxing	40c2e2e	BitVectorTestCase	1.00	0.75	1.00
abel533/Mapper	606f026	SpringAnnotationTest	1.00	1.00	1.00
dropwizard/metrics	3ac20d3	DefaultObjectNameFactoryTest	1.00	1.00	1.00
Activiti/Activiti	d5f49be	TimerScheduledListenerDelegateTest	1.00	1.00	1.00

dropwizard/metrics	bf85ca4	InstrumentedAppenderConfigTest	1.00	1.00	1.00
NationalSecurityAgency-/ghidra	e4372a3	ArrayStringableTest	1.00	1.00	1.00
apache/cassandra	5707323	PagingTest	1.00	1.00	1.00
eclipse-vertx/vert.x	6466066	ConcurrentCyclicSequenceTest	1.00	0.88	1.00
facebook/fresco	e0543ef	DownsampleUtilTest	1.00	1.00	1.00
signalapp/Signal-Android	f7b9942	CallParticipantListUpdateTest	1.00	1.00	1.00
apache/pulsar	82237d3	SinkTest	1.00	1.00	1.00
apache/zookeeper	de7c586	ReadOnlyZooKeeperServerTest	1.00	1.00	1.00
ben-manes/caffeine	ff58dbe	OSGiTest	1.00	0.50	1.00
eclipse-vertx/vert.x	9aa0e9f	IsolatingClassLoaderTest	0.80	0.80	1.00
alibaba/otter	2422bd8	NioUtilsTest	1.00	0.67	1.00
alibaba/druid	2db8321	CircleTest	NA	0.00	1.00
androidannotations-/androidannotations	4381942	RClassFinderTest	1.00	1.00	1.00
androidannotations-/androidannotations	17494d6	SharedPrefNamingTest	1.00	1.00	1.00
alibaba/spring-cloud-alibaba	290dfa7	RocketMQMessageConverterSupportTest	1.00	1.00	1.00
apolloconfig/apollo	cd77fd5	NamespaceBranchServiceTest	1.00	1.00	1.00
baomidou/mybatis-plus	16ea671	ReflectionKitTest	1.00	1.00	1.00

junit-team/junit4	a753708	JUnitCommandLineParseResultTest	1.00	1.00	0.83
ben-manes/caffeine	9796060	CoalescingBulkloaderTest	0.00	0.00	1.00
apache/skywalking	383a5bb	CounterWindowTest	NA	0.00	1.00
alibaba/canal	7aaee7b	JsonUtilsTest	1.00	1.00	1.00
apache/dolphinscheduler	14ec4a2	CommonUtilsTest	1.00	1.00	1.00
apache/skywalking	b8cf95d	SumPerMinFunctionTest	1.00	1.00	1.00
questdb/questdb	d8c9848	ToDateVCFunctionFactoryTest	1.00	1.00	1.00
thundernest/k-9	fdb8655	MailToTest	1.00	1.00	1.00
alibaba/easyexcel	0e546c3	FillTest	1.00	0.17	1.00
alibaba/druid	11cae3e	WallUpdateCheckTest	0.00	0.00	1.00
thingsboard/thingsboard	3112431	DefaultDeviceStateServiceTest	1.00	1.00	1.00
signalapp/Signal-Android	f7b9942	ParticipantCollectionTest	1.00	1.00	1.00
apolloconfig/apollo	ec2dabe	DefaultConfigFactoryTest	1.00	0.67	1.00
redis/jedis	618a850	DocumentTest	1.00	0.67	1.00
gocd/gocd	5c7c6c2	CreateConfigRepoCommandTest	1.00	1.00	1.00
baomidou/mybatis-plus	253574d	PageCacheTest	1.00	1.00	1.00
alibaba/fastjson	a45e6d5	StringTest_02	1.00	1.00	1.00
apache/pulsar	40f0438	ManagedCursorIndividual-DeletedMessagesTest	NA	0.00	1.00
baomidou/mybatis-plus	253574d	IDialectTest	1.00	1.00	1.00

ityouknow/spring-boot-examples	853b7df	MailServiceTest	1.00	0.80	1.00
seata/seata	edbf252	DataBaseSessionManagerTest	1.00	0.90	1.00
pagehelper/Mybatis-PageHelper	514308e	OffsetTest	1.00	1.00	1.00
apache/dolphinscheduler	49d581d	WebexTeamsAlertChannelFactoryTest	0.50	0.50	1.00
Activiti/Activiti	ee179a3	ELResolverReflectionBlockerDecoratorTest	1.00	1.00	1.00
apache/rocketmq	53baee7	PlainAccessControlFlowTest	1.00	1.00	1.00
dropwizard/dropwizard	9086577	JsonFormatterTest	1.00	0.50	1.00
dbeaver/dbeaver	f6bbe70	PostgreDialectFunctionsTest	1.00	1.00	1.00
json-path/JsonPath	8421159	DemoTest	1.00	1.00	1.00
TheAlgorithms/Java	ee2629c	PangramTest	1.00	1.00	1.00
bazelbuild/bazel	9556d0b	MemoizerTest	1.00	0.40	1.00
SonarSource/sonarqube	8f2b35b	EducationWith4LinkedCode-SnippetsSensorTest	1.00	1.00	1.00
perwendel/spark	9c5cab7	CollectionUtilsTest	1.00	1.00	1.00
alibaba/Sentinel	e5a62cb	SentinelDubboProviderFilterTest	1.00	1.00	1.00
apache/skywalking	439f843	ByteUtilTest	NA	0.00	1.00
bazelbuild/bazel	01a46f0	JavaInfoTest	1.00	1.00	1.00
apache/cassandra	4526b3f	ProfileLoadTest	NA	0.00	1.00

mockito/mockito	512ee39	StubbingReturnsSelfTest	1.00	1.00	1.00
apolloconfig/apollo	9b1cd8f	DefaultMessageProducerManagerTest	1.00	1.00	1.00
dropwizard/dropwizard	975ed69	LazyLoadingTest	0.67	0.67	1.00
apache/flink	6ff6978	PythonGroupWindow-AggregateJsonPlanTest	NA	0.00	1.00
dbeaver/dbeaver	f2efce7	SQLFormatterTokenizedTest	1.00	1.00	0.79
realm/realm-java	a1b984d	OsObjectStoreTests	1.00	1.00	1.00
abel533/Mapper	606f026	IdTest	1.00	1.00	1.00
bazelbuild/bazel	b194653	EditDuringBuildTest	1.00	1.00	1.00
dromara/hutool	d873b6e	CacheableSynthesizedAnnotation-AttributeProcessorTest	1.00	1.00	1.00
google/ExoPlayer	85bd080	RtspSessionTimingTest	1.00	1.00	1.00
TheAlgorithms/Java	00282ef	NextFitTest	0.80	0.80	1.00
alibaba/spring-cloud-alibaba	8d9790b	NacosDiscoveryClientConfigurationTest	0.50	0.50	1.00

Appendix B

Evaluation Results on Evaluation Set

Table B.1: Accuracy per Test Class on Evaluation Set

Repository	**Commit**	**Test Class**	accuracy_a	accuracy_g	accuracy_f
OpenRefine/OpenRefine	cfe9d37	RdfTripleImporterTests	1.00	1.00	1.00
json-path/JsonPath	af031cd	WithJsonPathTest	1.00	1.00	1.00
ben-manes/caffeine	4e54c96	AsyncTest	1.00	0.82	1.00
spring-projects/spring-framework	3b2b36d	AccessControlTests	0.96	0.96	1.00
thunderbird/thunderbird-android	942ecb5	MimeUtilityTest	1.00	1.00	1.00
neo4j/neo4j	6753f17	ByteUnitTest	1.00	0.69	1.00
apache/flink	c6997c9	ExceptionUtilsTest	0.85	0.79	1.00
signalapp/Singal-Android	f06817f	ProfileNameTest	0.93	0.93	1.00
dropwizard/dropwizard	f196ef9	JdbiTest	0.82	0.82	1.00
dromara/hutool	ddc4fdb	DbTest	1.00	1.00	1.00
SeleniumHQ/selenium	02744ac	ClickTest	0.92	0.82	0.95
apache/incubator-seata	80482ee	RpcContextTest	0.92	0.92	1.00
resilience4j/resilience4j	297dc71	VavrRetryTest	1.00	1.00	1.00
OpenFeign/feign	30814a1	ReactiveFeignIntegrationTest	0.82	0.64	1.00
redisson/redisson	cea6dc4	RedissonDequeTest	1.00	1.00	1.00

google/closure-compiler	6ac4656	TimelineTest	0.80	0.33	1.00
apache/dolphinscheduler	d46e465	DAGTest	1.00	1.00	1.00
NationalSecurityAgency-/ghidra	47373d2	FontValueTest	0.56	0.45	1.00
signalapp/Signal-Android	f06817f	ProfileNameTest	0.93	0.93	1.00
SeleniumHQ/selenium	43eb1e5	NetworkUtilsTest	0.57	0.40	1.00
google/ExoPlayer	053e14a	MatrixUtilsTest	1.00	1.00	1.00
halo-dev/halo	c400c85	MeterUtilsTest	1.00	1.00	1.00
redis/jedis	24653	StringValuesCommandsTestBase	1.00	0.96	1.00
jenkinsci/jenkins	b53e5ac	ActionableTest	1.00	0.86	1.00
eclipse-vertx/vert.x	694d9b2	AsynchronousLockTest	0.86	0.43	1.00
apache/dubbo	730695a	RegistryConfigTest	0.95	0.95	1.00
SonarSource/sonarqube	1977da9	GroupDaoIT	0.90	0.90	1.00
Netflix/zuul	7a375f5	HttpQueryParamsTest	1.00	0.90	1.00
thingsboard/thingsboard	c5a72ed	UserServiceTest	1.00	1.00	1.00
google/gson	04eb52d	MixedStreamTest	1.00	0.15	1.00
material-components/material-components-android	fd40fea	MaterialShapeDrawableTest	0.92	0.92	1.00

apache/shardingsphere-elasticjob	d413b44	JobRegistryTest	1.00	1.00	1.00
dbeaver/dbeaver	3e1d560	ArrayUtilsTest	1.00	1.00	1.00
facebook/buck	631d1a9	TypesTest	0.88	0.88	1.00
NationalSecurityAgency-/ghidra	79d8f16	RedBlackTreeTest	1.00	0.92	1.00
Activiti/Activiti	c9d398e	ApplicationUpgradeIT	1.00	0.70	1.00
facebook/litho	f8093d2	ScrollStateDetectorTest	1.00	0.83	1.00
apache/dolphinscheduler	a070aa9	WorkerGroupServiceTest	1.00	1.00	1.00
apolloconfig/apollo	6657a58	EnvTest	0.88	0.83	1.00
stanfordnlp/CoreNLP	48e748f	SentenceITest	0.75	0.55	1.00
google/closure-compiler	eb77dc5	QualifiedNameTest	0.87	0.87	1.00
pinpoint-apm/pinpoint	52abc30	StringUtilsTest	1.00	1.00	0.82
signalapp/Signal-Android	4656cf4	ExpirationUtilTest	1.00	1.00	1.00
halo-dev/halo	57fb644	QueryIndexViewImplTest	0.75	0.75	1.00
bumptech/glide	e9b8758	HttpUrlFetcherTest	0.58	0.54	0.88
grpc/grpc-java	8c4f4e0	ContextsTest	1.00	1.00	1.00
mybatis/mybatis-3	f42b1e7	ResourcesTest	0.29	0.29	1.00
ReactiveX/RxJava	1c3594c	MaybeRetryTest	1.00	0.40	1.00
apache/skywalking	383a5bb	ConnectUtilTestCase	0.89	0.89	1.00

questdb/questdb	d8c9848	BoolListTest	1.00	0.64	1.00
apache/shardingsphere-elasticjob	9425110	JobOperateAPIImplTest	1.00	1.00	1.00
apache/cassandra	34fa4e2	EmptyValuesTest	1.00	1.00	1.00
redisson/redisson	a4f4879	RedissonSetRxTest	1.00	0.94	1.00
realm/realm-java	a1b984d	RealmCollectionTests	1.00	1.00	1.00
facebook/fresco	b7778fc	PooledByteBufferInputStreamTest	1.00	0.77	1.00
facebook/buck	97bf736	AaptStepTest	1.00	1.00	1.00
apache/pulsar	82237d3	NamespaceNameTest	0.92	0.82	1.00
bumptech/glide	fab2aed	WideGamutTest	1.00	0.70	1.00
mockito/mockito	ea6ff8c	InvalidUsageTest	1.00	0.86	1.00
eugenp/tutorials	e14ea66	CSVUnitTest	1.00	1.00	1.00
alibaba/nacos	68b8e61	DiskUtilsTest	1.00	1.00	1.00
apache/kafka	2b26db0	ProducerPerformanceTest	1.00	0.62	1.00
Netflix/Hystrix	67fd60a	BasicObservableTest	0.60	0.60	1.00
google/auto	7d93867	MoreElementsTest	1.00	1.00	1.00
google/tsunami-security-scanner	2c8ac4f	TcsClientTest	1.00	0.91	1.00
antlr/antlr	4d5a6bf	TestUtils	1.00	1.00	1.00
xkcoding/spring-boot-demo	45bcd49	PersonRepositoryTest	0.75	0.30	1.00

zaproxy/zaproxy	ec3b940	StatisticsUnitTest	1.00	1.00	1.00
material-components/material-components-android	f4d0f56	UtcDatesTest	1.00	0.93	1.00
apache/zookeeper	bc1fc6d	QuorumServerTest	1.00	0.40	1.00
Netflix/eureka	9fa8437	EurekaEntityFunctionsTest	1.00	1.00	1.00
netty/netty	3efd964	ObjectUtilTest	1.00	1.00	1.00
swagger-api/swagger-core	063a5df	ReflectionUtilsTest	1.00	0.87	1.00
apache/zookeeper	3702a45	NetUtilsTest	0.36	0.33	1.00
google_guava/guava	41e0338	GraphPropertiesTest	1.00	1.00	1.00
google/auto	7373b98	BuilderRequiredPropertiesTest	1.00	1.00	1.00
zaproxy/zaproxy	ec3b940	KbUnitTest	1.00	1.00	1.00
java-native-access/jna	8cfb552	Dxva2Test	1.00	1.00	1.00
facebook/fresco	b7778fc	JfifTestUtilsTest	0.90	0.90	1.00
bazelbuild/bazel	cfef67d	ChunkerTest	0.89	0.67	1.00
apache/rocketmq	e0213fb	AclUtilsTest	1.00	0.87	1.00
questdb/questdb	e065400	RecordTest	1.00	1.00	1.00
MyCATApache/Mycat-Server	02f0ec1	UTF8StringSuite	0.96	0.92	1.00
checkstyle/checkstyle	8531fec	SarifLoggerTest	0.92	0.85	1.00
jhy/jsoup	4ea768d	StringUtilTest	1.00	0.75	1.00

dromara/hutool	acb4032	HtmlUtilTest	1.00	0.89	1.00
google/ExoPlayer	b0a3bc5	FlagSetTest	1.00	0.90	1.00
perwendel/spark	6945ea9	ServiceTest	1.00	0.79	1.00
signalapp/Signal-Server	7d483c7	KeysManagerTest	0.89	0.89	1.00
abel533/Mapper	606f026	TestLogicDelete	0.94	0.94	1.00
neo4j/neo4j	6753f17	LoaderTest	1.00	1.00	1.00
gocd/gocd	67fbd63	StageTest	1.00	0.94	1.00
spring-projects/spring-security	3e93b02	AclClassIdUtilsTests	1.00	1.00	1.00
eclipse-vertx/vert.x	ba568e5	HostAndPortTest	0.89	0.89	1.00
apache/dubbo	730695a	StackTest	1.00	0.90	1.00
square/retrofit	7de5ed9	MaybeTest	1.00	1.00	1.00
dropwizard/dropwizard	e3b7da7	HealthCheckManagerTest	1.00	1.00	1.00
apache/shardingsphere-elasticjob	066b556	TaskContextTest	1.00	0.54	1.00
redis/jedis	4d1de18	SetCommandsTest	1.00	0.95	1.00
codecentric/spring-boot-admin	d24edc2	InstanceDiscoveryListenerTest	1.00	1.00	1.00

Appendix C

CollectionUtilsTest Test Class

Figure C.1: CollectionUtilsTest Test Class

```
1  public class CollectionUtilsTest {
2   @Test
3   @FocalMethod("IsEmpty")
4   @State("CollectionIsEmpty")
5   @ExpectedResult("ReturnTrue")
6   public void testIsEmpty_whenCollectionIsEmpty_thenReturnTrue() {
7    Collection<Object> testCollection = new ArrayList<>();
8    assertTrue("Should return true because collection is empty", CollectionUtils.isEmpty(testCollection));
9   }
10  @Test
11  @FocalMethod("IsEmpty")
12  @State("CollectionIsNotEmpty")
13  @ExpectedResult("ReturnFalse")
14  public void testIsEmpty_whenCollectionIsNotEmpty_thenReturnFalse() {
15   Collection<Integer> testCollection = new ArrayList<>();
16   testCollection.add(1);
17   testCollection.add(2);
18   assertFalse("Should return false because collection is not empty", CollectionUtils.isEmpty(testCollection));
19  }
20  @Test
21  @FocalMethod("IsEmpty")
22  @State("CollectionIsNull")
23  @ExpectedResult("ReturnTrue")
24  public void testIsEmpty_whenCollectionIsNull_thenReturnTrue() {
25   Collection<Integer> testCollection = null;
26   assertTrue("Should return true because collection is null", CollectionUtils.isEmpty(testCollection));
27  }
28  @Test
29  @FocalMethod("IsNotEmpty")
30  @State("CollectionIsEmpty")
31  @ExpectedResult("ReturnFalse")
32  public void testIsNotEmpty_whenCollectionIsEmpty_thenReturnFalse() {
33   Collection<Object> testCollection = new ArrayList<>();
34   assertFalse("Should return false because collection is empty", CollectionUtils.isNotEmpty(testCollection));
35  }
36  @Test
37  @FocalMethod("IsNotEmpty")
38  @State("CollectionIsNotEmpty")
39  @ExpectedResult("ReturnTrue")
40  public void testIsNotEmpty_whenCollectionIsNotEmpty_thenReturnTrue() {
41   Collection<Integer> testCollection = new ArrayList<>();
42   testCollection.add(1);
43   testCollection.add(2);
44   assertTrue("Should return true because collection is not empty", CollectionUtils.isNotEmpty(testCollection));
45  }
46  @Test
47  @FocalMethod("IsNotEmpty")
48  @State("CollectionIsNull")
49  @ExpectedResult("ReturnFalse")
50  public void testIsNotEmpty_whenCollectionIsNull_thenReturnFalse() {
51   Collection<Object> testCollection = null;
52   assertFalse("Should return false because collection is null", CollectionUtils.isNotEmpty(testCollection));
53  }
54 }
```

Appendix D

CollectionUtilsTest Test Class Classified by Default Strategy

Figure D.1: CollectionUtilsTest Test Class Classified by Default Strategy.

```
1  public class CollectionUtilsTest {
2   @Nested
3   class IsNotEmpty {
4    @Nested
5    class ReturnFalse {
6     @Test
7     @FocalMethod("IsNotEmpty")
8     @State("CollectionIsEmpty")
9     @ExpectedResult("ReturnFalse")
10    public void testIsNotEmpty_whenCollectionIsEmpty_thenReturnFalse() {
11     ...
12    }
13    @Test
14    @FocalMethod("IsNotEmpty")
15    @State("CollectionIsNull")
16    @ExpectedResult("ReturnFalse")
17    public void testIsNotEmpty_whenCollectionIsNull_thenReturnFalse() {
18     ...
19   }
20   @Test
21   @FocalMethod("IsNotEmpty")
22   @State("CollectionIsNotEmpty")
23   @ExpectedResult("ReturnTrue")
24   public void testIsNotEmpty_whenCollectionIsNotEmpty_thenReturnTrue() {
25    ...
26   }
27  }
28  @Nested
29  class IsEmpty {
30   @Nested
31   class ReturnTrue {
32    @Test
33    @FocalMethod("IsEmpty")
34    @State("CollectionIsNull")
35    @ExpectedResult("ReturnTrue")
36    public void testIsEmpty_whenCollectionIsNull_thenReturnTrue() {
37     ...
38    }
39    @Test
40    @FocalMethod("IsEmpty")
41    @State("CollectionIsEmpty")
42    @ExpectedResult("ReturnTrue")
43    public void testIsEmpty_whenCollectionIsEmpty_thenReturnTrue() {
44     ...
45    }
46   }
47   @Test
48   @FocalMethod("IsEmpty")
49   @State("CollectionIsNotEmpty")
50   @ExpectedResult("ReturnFalse")
51   public void testIsEmpty_whenCollectionIsNotEmpty_thenReturnFalse() {
52    ...
53   }
54  }
55 }
```

Appendix E

CollectionUtilsTest Test Class Classified by Result Fragment

Figure E.1: CollectionUtilsTest Test Class Classified by Result Fragment.

```
1  public class CollectionUtilsTest {
2   @Nested
3   class ReturnFalse {
4    @Nested
5    class IsNotEmpty {
6     @Test
7     @FocalMethod("IsNotEmpty")
8     @State("CollectionIsEmpty")
9     @ExpectedResult("ReturnFalse")
10    public void testIsNotEmpty_whenCollectionIsEmpty_thenReturnFalse() {
11     ...
12    }
13    @Test
14    @FocalMethod("IsNotEmpty")
15    @State("CollectionIsNull")
16    @ExpectedResult("ReturnFalse")
17    public void testIsNotEmpty_whenCollectionIsNull_thenReturnFalse() {
18     ...
19    }
20   }
21   @Test
22   @FocalMethod("IsEmpty")
23   @State("CollectionIsNotEmpty")
24   @ExpectedResult("ReturnFalse")
25   public void testIsEmpty_whenCollectionIsNotEmpty_thenReturnFalse(){
26    ...
27   }
28  }
29  @Nested
30  class ReturnTrue {
31   @Nested
32   class IsEmpty {
33    @Test
34    @FocalMethod("IsEmpty")
35    @State("CollectionIsNull")
36    @ExpectedResult("ReturnTrue")
37    public void testIsEmpty_whenCollectionIsNull_thenReturnTrue() {
38     ...
39    }
40    @Test
41    @FocalMethod("IsEmpty")
42    @State("CollectionIsEmpty")
43    @ExpectedResult("ReturnTrue")
44    public void testIsEmpty_whenCollectionIsEmpty_thenReturnTrue() {
45     ...
46    }
47   }
48   @Test
49   @FocalMethod("IsNotEmpty")
50   @State("CollectionIsNotEmpty")
51   @ExpectedResult("ReturnTrue")
52   public void testIsNotEmpty_whenCollectionIsNotEmpty_thenReturnTrue() {
53    ...
54   }
55  }
56 }
```

Appendix F

CollectionUtilsTest Test Class Classified by Method Fragment

Figure F.1: CollectionUtilsTest Test Class Classified by Method Fragment.

```
1  public class CollectionUtilsTest {
2   @Nested
3   class IsNotEmpty {
4    @Nested
5    class ReturnFalse {
6     @Test
7     @FocalMethod("IsNotEmpty")
8     @State("CollectionIsEmpty")
9     @ExpectedResult("ReturnFalse")
10    public void testIsNotEmpty_whenCollectionIsEmpty_thenReturnFalse() {
11     ...
12    }
13    @Test
14    @FocalMethod("IsNotEmpty")
15    @State("CollectionIsNull")
16    @ExpectedResult("ReturnFalse")
17    public void testIsNotEmpty_whenCollectionIsNull_thenReturnFalse() {
18     ...
19    }
20   }
21   @Test
22   @FocalMethod("IsNotEmpty")
23   @State("CollectionIsNotEmpty")
24   @ExpectedResult("ReturnTrue")
25   public void testIsNotEmpty_whenCollectionIsNotEmpty_thenReturnTrue() {
26    ...
27   }
28  }
29  @Nested
30  class IsEmpty {
31   @Nested
32   class ReturnTrue {
33    @Test
34    @FocalMethod("IsEmpty")
35    @State("CollectionIsNull")
36    @ExpectedResult("ReturnTrue")
37    public void testIsEmpty_whenCollectionIsNull_thenReturnTrue() {
38     ...
39    }
40    @Test
41    @FocalMethod("IsEmpty")
42    @State("CollectionIsEmpty")
43    @ExpectedResult("ReturnTrue")
44    public void testIsEmpty_whenCollectionIsEmpty_thenReturnTrue() {
45     ...
46    }
47   }
48   @Test
49   @FocalMethod("IsEmpty")
50   @State("CollectionIsNotEmpty")
51   @ExpectedResult("ReturnFalse")
52   public void testIsEmpty_whenCollectionIsNotEmpty_thenReturnFalse() {
53    ...
54   }
55  }
56 }
```

Appendix G

CollectionUtilsTest Test Class Classified by State Fragment

Figure G.1: CollectionUtilsTest Test Class Classified by State Fragment.

```
1  public class CollectionUtilsTest {
2   @Nested
3   class CollectionIsEmpty {
4    @Test
5    @FocalMethod(”IsNotEmpty”)
6    @State(”CollectionIsEmpty”)
7    @ExpectedResult(”ReturnFalse”)
8    public void testIsNotEmpty_whenCollectionIsEmpty_thenReturnFalse() {
9     ...
10   }
11   @Test
12   @FocalMethod(”IsEmpty”)
13   @State(s”CollectionIsEmpty”)
14   @ExpectedResult(”ReturnTrue”)
15   public void testIsEmpty_whenCollectionIsEmpty_thenReturnTrue() {
16    ...
17   }
18  }
19  @Nested
20  class CollectionIsNotEmpty {
21   @Test
22   @FocalMethod(”IsEmpty”)
23   @State(”CollectionIsNotEmpty”)
24   @ExpectedResult(”ReturnFalse”)
25   public void testIsEmpty_whenCollectionIsNotEmpty_thenReturnFalse() {
26    ...
27   }
28   @Test
29   @FocalMethod(”IsNotEmpty”)
30   @State(”CollectionIsNotEmpty”)
31   @ExpectedResult(”ReturnTrue”)
32   public void testIsNotEmpty_whenCollectionIsNotEmpty_thenReturnTrue() {
33    ...
34   }
35  }
36  @Nested
37  class CollectionIsNull {
38   @Test
39   @FocalMethod(”IsNotEmpty”)
40   @State(”CollectionIsNull”)
41   @ExpectedResult(”ReturnFalse”)
42   public void testIsNotEmpty_whenCollectionIsNull_thenReturnFalse() {
43    ...
44   }
45   @Test
46   @FocalMethod(”IsEmpty”)
47   @State(”CollectionIsNull”)
48   @ExpectedResult(”ReturnTrue”)
49   public void testIsEmpty_whenCollectionIsNull_thenReturnTrue() {
50    ...
51   }
52  }
53 }
```