Aiding Comprehension of Unit Test Cases and Test Suites with Stereotype-based Tagging

Boyang Li
College of William and Mary
Williamsburg, VA

Christopher Vendome
College of William and Mary
Williamsburg, VA

Mario Linares-Vásquez
Universidad de los Andes
Bogotá, Colombia

Denys Poshyvanyk
College of William and Mary
Williamsburg, VA

ABSTRACT

Techniques to automatically identify the stereotypes of different software artifacts (e.g., classes, methods, commits) were previously presented. Those approaches utilized the techniques to support comprehension of software artifacts, but those stereotype-based approaches were not designed to consider the structure and purpose of unit tests, which are widely used in software development to increase the quality of source code. Moreover, unit tests are different than production code, since they are designed and written by following different principles and workflows.

In this paper, we present a novel approach, called TeStereo, for automated tagging of methods in unit tests. The tagging is based on an original catalog of stereotypes that we have designed to improve the comprehension and navigation of unit tests in a large test suite. The stereotype tags are automatically selected by using static control-flow, data-flow, and API call based analyses. To evaluate the benefits of the stereotypes and the tagging reports, we conducted a study with 46 students and another survey with 25 Apache developers to (i) validate the accuracy of the inferred stereotypes, (ii) measure the usefulness of the stereotypes when writing/understanding unit tests, and (iii) collect feedback on the usefulness of the generated tagging reports.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**;

KEYWORDS

Unit test cases, program comprehension, maintaining software

ACM Reference Format:
Boyang Li, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. 2018. Aiding Comprehension of Unit Test Cases and Test Suites with Stereotype-based Tagging . In *ICPC '18: ICPC '18: 26th IEEE/ACM International Confernece on Program Comprehension , May 27–28, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3196321.3196339

ICPC '18, May 27–28, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5714-2/18/05...$15.00
https://doi.org/10.1145/3196321.3196339

1 INTRODUCTION

Unit testing is considered to be one of the most popular automated techniques to detect bugs in software, perform regression testing, and, in general, to write better code [32, 44]. In fact, unit testing is (i) the foundation for approaches such as Test First Development (TFD) [16] and Test-Driven Development (TDD) [11, 15], (ii) one of the required practices in agile methods such as XP [16], and (iii) has inspired other approaches such as Behavior-Driven Development (BDD) [50]. In general, unit testing requires writing "test code" by relying on APIs such as the XUnit family [2, 4, 6] or Mock-based APIs such as Mockito [3] and JMockit [1].

Besides the usage of specific APIs for testing purposes, unit test code includes calls to the system under test, underlying APIs (e.g., the Java API), and programming structures (e.g., loops and conditionals), similarly to production code (i.e., non-test code). Therefore, unit test code can also exhibit issues such as bad smells [14, 63], poor readability, and textual/syntactic characteristics that impact program understanding. In addition, despite the existence of tools for automatic generation of unit test code [12, 27–29, 51, 55], automatically generated test cases (i.e., unit tests) are difficult to understand and maintain [52]. As a response to the aforementioned issues, several guidelines for writing and refactoring unit tests have been proposed [32, 44, 63].

To bridge this gap, this work proposes a novel automated catalog of stereotypes for methods in unit tests; the catalog was designed with the goal of improving the comprehension of unit tests and navigability of large test suites. The approach is a complementary technique to the existing approaches [36, 40, 52], which generate detailed summaries for each test method without considering method stereotypes at the test suite level.

While code stereotypes reflect high-level descriptions of the roles of a code unit (e.g., a class or a method) and have been defined before for production code [7, 22, 24], our catalog is first to capture unit test case specific stereotypes. Based on the catalog, the paper also presents an approach, coined as *TeStereo*, for automatically tagging methods in unit tests according to the stereotypes to which they belong. *TeStereo* generates browsable documentation for a test suite (e.g., an html-based report), which includes navigation features, source code, and the unit tests tags. *TeStereo* generates the stereotypes at unit test method level by identifying (i) any API call or references to the JUnit API (i.e., assertions, assumptions, fails, annotations), (ii) inter-procedural calls to the methods in the same unit test and external methods (i.e., internal methods or external APIs), and (iii) control/data-flows related to any method call.

To validate the accuracy and usefulness of test case stereotypes and *TeStereo*'s reports, we designed and conducted three experiments based on 231 Apache projects as well as 210 test case methods, which were selected from the Apache projects by using a sampling procedure aimed at getting a diverse set of methods in terms of size, number, and type of stereotypes detected in the methods (Section 4.2). In these projects, *TeStereo* detected an average of 1,577 unit test stereotypes per system, which had an average of 5.90 unit test methods per test class (total of 168,987 unit test methods from 28,644 unit test classes). When considering the total dataset, the prevalence of any single stereotype ranged from 482 to 67,474 instance of the stereotype. In addition, we surveyed 25 Apache developers regarding their impressions and feedback on *TeStereo* 's reports. Our experimental results show that (i) *TeStereo* achieves very high precision and recall for detecting the proposed unit test stereotypes; (ii) the proposed stereotypes improve comprehension of unit test cases during maintenance tasks; and (iii) most of the developers agreed that stereotypes and reports are useful for test case comprehension.

In summary, this paper makes the following contributions: (i) a catalog of 21 stereotypes for methods in unit tests that extensively consider the JUnit API, external/internal inter-procedure calls, and control/data-flows in unit test methods; (ii) a static analysis-based approach for identifying unit test stereotypes; (iii) an open source tool that implements the proposed approach and generates stereotype-based reports documenting test suites [1]; and (iv) an extensive online appendix [5] that includes test-case related statistics of the analyzed Apache projects, the *TeStereo* reports of the 231 Apache projects, and the detailed data collected during the studies.

2 UNIT TEST CASE STEREOTYPES

In this section, we provide some background on stereotypes and describe the catalog of stereotypes that we have designed for unit tests methods.

Code stereotypes reflect roles of program entities (e.g., a class or a method) in a system, and those roles can be used for maintenance tasks such as design recovery, feature location, program comprehension, and pattern/anti-pattern detection [8, 23, 24]. Although detecting stereotypes is a task that can be done manually, it is prohibitively time-consuming in the case of a large software system [23, 24]. Therefore, automated approaches have been proposed to detect stereotypes for entities such as classes, methods, and commits [21, 23, 24]. However, the previously proposed catalog of stereotypes were not designed to consider the structure and purpose of unit tests; unit test cases are different than other artifacts, since unit tests are designed by following different principles and workflow than non-test code [19, 32, 44].

Consequently, we designed a catalog of 21 stereotypes for unit test methods (Table 1), under the hypothesis that stereotypes could help developers/testers to understand the responsibilities of unit tests within a test suite. Also, stereotypes may reflect a high-level description of the role of a unit test case. For instance, stereotypes such as *"Exception verifier"*, *"Iterative verifier"*, and *"Empty test"* are descriptive "tags" that can help developers to (i) identify the general purpose of the methods without exploring the source code, and (ii) navigate large test suites. Therefore, the stereotypes can be used as "tags" that annotate the unit test directly in the IDE, or in external documentation (e.g., an html report). The tags can be assist navigation/classification of test methods in large test suites. For example, it is time-consuming to manually identify test initializers that are also verifiers in a project like `Openejb` with 317 test classes and 1641 test method tags.

[1]https://github.com/boyangwm/TestStereotype/

```
TestCleaner EmptyTester
@After public void tearDown() throws Exception { }
```

Figure 1: *Test Cleaner* and *Empty Tester* method from `SessionTrackerCheckTest` unit test in `Zookeeper`.

```
TestInitializer InternalCallVerifier
NullVerifier IdentityVerifier HybridVerifier
@Before @Override public void setUp() throws Exception {
  super.setUp();
  this.tomcat=getTomcatInstance();
  this.context=this.tomcat.addContext("/weaving",WEBAPP_DOC_BASE);
  this.tomcat.start();
  ClassLoader
       loader=this.context.getLoader().getClassLoader();
  assertNotNull("The class loader should not be
       null.",loader);
  assertSame("The class loader is not
       correct.",WebappClassLoader.class,loader.getClass());
  this.loader=(WebappClassLoader)loader;}
```

Figure 2: *Test initializer* method (from `TestWebappClassLoaderWeaving` unit test in `Tomcat`) with other stereotypes detected by *TeStereo*.

Note that the catalog we propose in this paper focuses on unit tests relying on the JUnit API; we based this decision on the fact that in a sample of 381,161 open source systems from GitHub that we analyzed (by relying on a mining-based study) only 134 of the systems used a mock-style-only APIs while 8,556 systems used JUnit-only APIs.

The full list of stereotypes are described with their explanations in the following subsections and in Table 1, where we list the stereotypes, a brief description, and the rules used for their detection. The stereotypes were defined by considering how values or objects are verified in a unit test case, the responsibilities of the test case, and the data/control-flows in the unit test case. Therefore, we categorized the stereotypes in two categories that reflect the usage of the JUnit API, and the data/control-flows in the methods. Note that the categories and stereotypes are not mutually exclusive, because our goal is to provide developers/testers with a mechanism to navigate large test suites or identify unit test methods with multiple purposes. For example, the method in Figure 1 is an *"Empty tester"* and *"Test Cleaner"*; assuming that the methods are annotated (in someway) with the tags (i.e., stereotypes), developers/testers can locate all unimplemented methods in the test suite (i.e., the empty testers), which will also be executed the last during the test unit execution (i.e., the test cleaners). Another example of potential usage of the tags, is detecting strange or potentially smelly methods, such as the *"Test initializer"* (i.e., a method with the `@Before` annotation) method depicted in Figure 2, which has other tags such as *"Internal Call Verifier"*, and *"Null Verifier"*; we think this is a smelly methods because test initializer are not supposed to have assertions.

2.1 JUnit API-based Stereotypes

Assertions in the JUnit API have well-defined semantics that can be used to automatically infer or document the purpose of a test case [40, 52]. However, besides assertions for validating logical conditions between expected and real results (e.g., `assertEquals(int,int)`),

Table 1: Proposed Stereotypes for Methods in Unit Test Cases.

	Type	Description	Rules
JUnit API-Based	Boolean verifier	Verifies boolean conditions	Contains `assertTrue` || Contains `assertFalse`
	Null verifier	Verifies whether objects are null	Contains `assertNull` || Contains `assertNotNull`
	Equality verifier	Verifies whether objects/variables are equal to an expected value	Contains `assertEquals` || Contains `assertArrayEquals` || Contains `assertNotEquals`
	Identity verifier	Verifies whether two objects/variables refer to the same object/variable	Contains `assertSame` || Contains `assertNotSame`
	Utility verifier	Verifies (un)successful execution of the test case by reporting explicitly a failure	Contains `fail`
	Exception verifier	Verifies that exceptions are thrown during the test case execution	Has `Expected` attribute with the value class `Exception` or classes inherited from `Exception`
	Condition Matcher	Verifies logic rules using matcher-style statements	Contains `assertThat`
	Assumption setter	Sets implicit assumptions	Contains `assumeThat` || Contains `assumeTrue`
	Test initializer	Allocates resources before the execution of the test cases	Has annotation `@Before` || Has annotation `@BeforeClass`
	Test cleaner	Releases resources used by the test cases	Has annotation `@After` || Has annotation `@AfterClass`
	Logger	Invokes logging operations	Calls functions in `PrintStream` class || Calls functions in `Logger` class
	Ignored method	Is not executed with the test suite	Has annotation `@Ignore`
	Hybrid verifier	Contains more than one JUnit-based stereotype	Number of matched JUnit-based stereotype > 1
	Unclassified	Is not categorized by any of the available tags	Number of matched JUnit-based stereotype == 0
C/D-Flow Based	Branch verifier	Verifies assertions inside branch conditions	Number of assertions within branch conditions > 0
	Iterative verifier	Verifies assertions in iterations	Number of assertions within iterations > 0
	Public field verifier	Verifies values related to public fields	Actual values in assertions are from public field accesses
	API utility verifier	Verifies values of variables related to API calls (External libraries)	Actual values in assertions are values of objects/variables related to API calls
	Internal call verifier	Verifies values of variables related to AUT calls	Actual values in assertions are values of objects/variables related to AUT calls
	Execution tester	Executes/invoke methods but no assertions are verified	Number of assertions == 0 && number of function calls > 0
	Empty tester	Is an empty test case	Number of lines of codes in method body == 0

```
1 @Test public void existingConfigurationReturned(){
2   Configuration conf=new Configuration(false);
3   conf.set("foo","bar");
4   Configuration conf2=CredentialProviderFactoryShim
5       .getConfiguration(conf,"jceks:///file/accumulo.jceks");
6   Assert.assertSame(conf,conf2);
7   Assert.assertEquals("bar",conf.get("foo"));}
```

Figure 3: Source code of the `existingConfigurationReturned` unit test method in the `Apache-accumulo`.

JUnit provides other APIs for defining assumptions, expected exceptions, matching conditions, explicit declaration of failures, fixture setters, and cleaners, which have not been considered in prior work in automatic generation of documentation. Our catalog includes stereotypes for each one of those cases, because those APIs can reflect different purposes and responsibilities of the methods using the unit testing APIs.

The stereotypes in this category are detected by (i) building an Abstract Syntax Tree (AST) for each method, (ii) looking for invocations to methods and annotations with the same signature from the JUnit API, and (iii) using the set of rules listed in Table 1. For instance, Figure 3 is an example of *Identity verifier* and *Equality verifier*. The difference between those two stereotypes is that the former focuses on testing whether two objects are the same reference, while the latter focuses on verifying that the objects are the same (by using the `equals` method). In Figure 3, the assertion (in line 6) is an identity assert, since the function call `assertSame` asserts that `conf2` should be the same object reference as `conf` (indicated as *Identity verifier*). In line 7, `assertEquals` asserts that the returned string, by calling `conf.get("foo")`, is equal to `"bar"` (indicated as *Equality verifier*).

In addition to the API-based stereotypes, we also defined two stereotypes for cases in which a unit test case contains more than one JUnit-based stereotype (i.e., *Hybrid verifier*), and cases where *TeStereo* was not able to detect any of the stereotypes (i.e., *Unclassified*). Because of space limitations, we do not show examples for

```
@Test public void testConstructorMixedStyle(){
      Path p = new Path(project, "\\a;\\b:/c");
      String[] l = p.list();
      assertEquals("three items, mixed style", 3,
          l.length);
      if (isUnixStyle) {
         assertEquals("/a", l[0]);
         assertEquals("/b", l[1]);
         assertEquals("/c", l[2]);
      } else if (isNetWare) {
         assertEquals("\\a", l[0]);
         assertEquals("\\b", l[1]);
         assertEquals("\\c", l[2]);
      } else { ... }}
```

Figure 4: Source code of the `testConstructorMixedStyle` unit test method in the `Apache-ant` system.

all the stereotypes; however, more examples can be found in our online appendix [5].

2.2 Data-/Control-flow Based Stereotypes

The API-based stereotypes (Section 2.1) describe the purpose of the API invocations; however, those stereotypes neither describe how the unit test cases use the APIs nor from where the examined data (i.e., arguments to the API methods) originates. Therefore, we extended the list of stereotypes with a second category based on data/control-flow analyses, because these analyses can capture the information missing in API-based stereotypes.

Using control-flow information, we defined two stereotypes for reporting whether the JUnit API methods are invoked inside a loop (i.e., an *Iterative Verifier*) or inside conditional branches (i.e., a *Branch Verifier*). For example, the unit test case in Figure 4 is a *Branch Verifier*, which verifies that the constructor of class `Path` is able to handle mixed system path styles (i.e., Unix, NetWare, etc.).

Using data-flow information, we defined stereotypes that describe when the arguments to the JUnit API calls are from (i) accesses to public fields (*Public Field Verifier*), (ii) API calls different to JUnit (*API Utility Verifier*), or (iii) calls to the application under test (AUT) (*Internal Call Verifier*). For example, the unit test case

```
1  @Test public void testRead() throws Exception {
2    testConfigure();
3    Locator locator=new Locator("evar1",_pid,_iid);
4    Value value=new Value(locator,_el1,null);
5    value=_engine.writeValue(_varType,value);
6    Value readVal=_engine.readValue(_varType,value.locator);
7    assertEquals(_iid,readVal.locator.iid);
8    assertEquals(_pid,readVal.locator.pid);
9    assertEquals(2,DOMUtils.countKids((Element)
10     readVal.locator.reference,Node.ELEMENT_NODE));}
```

Figure 5: Source code of the `testRead` **unit test method in the** `ode` **system.**

in Figure 5 is a *Public Field Verifier*, which verifies the attributes in `readVal.locator` have the expected values. The data flow is from line 4 where the `value` object is created, to line 6 where the public field `value.locator` is accessed and used as an argument for a method invocation that is assigned to `readVal`. There is also a stereotype for methods in unit tests that do not verify assertions but invoke internal or external methods (*Execution Tester*). Finally, we included a stereotype that describes empty methods (*Empty tester*); developers/testers can use this type of stereotype to easily locate unimplemented methods in the test suite. Our online appendix[5] has examples of each stereotype.

3 DOCUMENTING UNIT TEST CASES WITH TESTEREO

TeStereo is an approach for automatically documenting unit test suites that (i) tags methods in test cases by using the stereotypes and rules defined in Section 2, and (ii) builds an html-formatted report that includes the tags, source code, and navigation features, such as filters and navigation trees. In addition, for each method in a unit test, *TeStereo* generates a summary based on the descriptions of each stereotype. *TeStereo* is a novel approach that combines static analysis and code summarization techniques in order to automatically generate tags and natural language-based descriptions aiming at concisely documenting the purpose of test cases.

TeStereo can be summarized in the following workflow:

1. Test case detection. The starting point of *TeStereo* is the source code of the system (including the test cases). *TeStereo* first analyzes the source code to identify all the unit test cases by detecting the methods in the source code that are annotated with `@Test`,`@Before`, `@BeforeClass`, `@After`, `@Afterclass` and `@Ignore`;

2. JUnit API call detection. The source code methods identified as test cases are then analyzed statically by scanning and detecting invocations to annotations and methods from the JUnit API;

3. Data/Control-flow analyses. Data-flow dependencies between the JUnit API calls and the variables defined in the analyzed method are identified by performing static backward slicing [35]; in addition, the collected references to the API calls are augmented with boolean flags reporting whether the calls are made inside loops or conditional branches. *TeStereo* performs a lightweight over-approximate analysis for each argument v in an JUnit API call to compute all potential paths (including internal function calls, Java API calls, and public field accesses) that may influence the value of v by using backward slicing [35]. Although *TeStereo* does not track any branch conditions in the unit test case (some paths may not be executed with certain inputs), the over-approximation guarantees that potential slices are not missed in the backward slicing relationships;

4. Stereotype detection. *TeStereo* uses the data collected in the previous steps, and then applies the rules listed in Table 1 to classify the unit tests into defined stereotype categories;

5. Report generation. Finally, each method is documented (as in Figure 7) and all the method level documents are organized in an html-based report. We encourage an interested reader to see the reports generated for 231 Apache projects in our online appendix [5].

4 EMPIRICAL STUDY

We conducted an empirical study aimed at (i) validating the accuracy of *TeStereo*-generated test case stereotypes, and (ii) the usefulness of the stereotypes and the reports for supporting evolution and maintenance of unit tests. We relied on CS students, researchers, and the original developers of Apache projects to perform the study. In particular, the *context* of the study encompasses 210 methods randomly selected from unit tests in 231 Apache projects, 231 *TeStereo* reports, 420 manually generated summaries, 25 Apache developers, and 46 students and researchers. The *perspective* is of researchers interested in techniques and tools for improving program comprehension and automatic documentation of unit tests.

4.1 Research Questions

In the context of our study, we investigated the following three research questions (RQs):

RQ$_1$: *What is TeStereo's accuracy for identifying unit test stereotypes?* Before using the stereotypes in experiments with students, researchers, and practitioners, we wanted to measure *TeStereo*'s accuracy in terms of precision and recall. The rules used for stereotype identification are based on static detection of API calls and data/control flow analyses. Therefore, with **RQ$_1$**, we aim at identifying whether *TeStereo* generates false positives or false negatives and the reasons behind them.

RQ$_2$: *Do the proposed stereotypes improve comprehension of tests cases (i.e., methods in test units)?* The main goal of method stereotypes is to describe the general purposes of the test cases in a unit test. Our hypothesis is that the proposed stereotypes should help developers in evolution and maintenance tasks that require program comprehension of unit tests. **RQ$_2$** aims at testing the hypothesis, in particular, when using the task of manually generating summaries/descriptions for methods in unit tests (with and without stereotypes) as a reference.

RQ$_3$: *What are the developers' perspectives of the TeStereo-based reports for systems in which they contributed?* *TeStereo* not only identifies test stereotypes at method level, but also generates html reports (i.e., documentation) that includes source code, stereotypes, short stereotype-based summaries, and navigation features. Thus, **RQ$_3$** aims at validating with practitioners (i) if the stereotypes and reports are useful for software-related tasks, (ii) what features in the reports are the most useful, and (iii) what improvements should be done to the reports if any.

The three RQs are complementary for *TeStereo*'s evaluation. **RQ$_1$** focuses on the quality of stereotype identification; we asked graduate CS students from a research university to manually identify the stereotypes on a sample of methods from unit tests; then, we computed micro and macro precision and recall metrics [57] between the gold-set generated by the students and the stereotypes identified by *TeStereo* on the same sample. With $\mathbf{RQ}_1$, we also manually checked the cases in which *TeStereo* was not able to correctly identify the stereotypes, and then improved our implementation. $\mathbf{RQ}_2$ focuses on the usefulness of method stereotypes in unit tests; thus, we first asked students and researchers to write summaries of the methods (when reading the code with and without stereotypes). Then, giving the source code and manually written summaries, we asked another group of students and researchers to evaluate the summaries in terms of completeness, conciseness, and expressiveness [17, 40, 46, 58]. Note that there is no overlap among the participants assigned to $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$. Finally, $\mathbf{RQ}_3$ focuses on the usefulness of stereotypes and reports from the practitioners' perspective.

4.2 Context Selection

For the three RQs, we used the population of unit tests included in 231 Apache projects with source code available at GitHub. The list of projects is provided in our online appendix [5]. Our preference for Apache projects is motivated by the fact that they have been widely used in previous studies performed by the research community [13, 45, 54], and unit tests in these projects are highly diverse in terms of method stereotypes, methods size (i.e., LOC), and the number of stereotypes. In the 231 projects, we detected a total of 27,923 unit tests, which account for 164,373 methods. Figures describing the diversity of the unit tests in 231 projects are in our online appendix [5]. On average, the methods have 14.67 LOC (median=10), the first quartile $Q1$ is 6 LOC, and the third quartile $Q3$ is 18 LOC. Concerning the number of stereotypes per system, on average, *TeStereo* identified 1,577 stereotypes in the unit tests (median=489). All 231 Apache projects exhibited at least 482 instances of each stereotype in the unit test methods, having *EqualityVerifier* as the most frequent method stereotype (64,474 instances). Finally, most of the methods (i.e., 73,906) have only one stereotype; however, there are cases with more than one stereotype, having a limit of 92 methods with 9 stereotypes each. In summary, the sample of Apache projects is diverse in terms of size of methods in the unit tests and the identified stereotypes (all 21 stereotypes were widely identified). Hereinafter, we will refer to the set of all the unit tests in 231 Apache projects as UT_{Apache}.

Because of the large set of unit test methods in UT_{Apache} (i.e., 164,373 methods), we sampled a smaller set of methods that could be evaluated during our experiments; we call this set M_{sample}, which is composed of 210 methods systematically sampled from the methods in UT_{Apache}. The reason for choosing 210 methods is that we wanted to have in the sample at least 10 methods representative of each stereotype (21 stereotypes ×10 methods = 210). Subsequently, given the target size for the sample, we designed a systematic sampling process looking for diversity in terms of not only stereotypes and the number of stereotypes per method but also selecting methods with a "representative" size (by "representative" we mean that the size is defined by the 50% of the original population). Therefore, we selected methods with LOC between $Q1 = 6$ and $Q3 = 18$. Consequently, after selecting only the methods with LOC $\in [Q1, Q3]$, we sampled them in buckets

Algorithm 1: Sampling procedure of methods from the whole set of unit test in the 231 Apache projects.

```
   Input: B⟨stereotype⟩, B(2)⟨stereotype⟩, B⟨n,stereotype⟩
   Output: M_sample
1  begin
2      N = [1..9], ST = ["Logger"..."Unclassified"];
3      M_sample = ∅; Counter⟨stereotype⟩ = ∅;
4      foreach ⟨n, stereotype⟩ ∈ N × ST do
5          m = pickRandomFrom(B⟨n,stereotype⟩);
6          if m ∉ M_sample then
7              M_sample.add(m);
8              Counter⟨stereotype⟩ ++;
9      while |M_sample| < 210 do
10         foreach stereotype ∈ ST do
11             if Counter⟨stereotype⟩ < 10 then
12                 selected = FALSE;
13                 m = pickRandomFrom(B⟨stereotype⟩);
14                 if m ∉ M_sample then
15                     selected = TRUE;
16                 if !selected then
17                     m = pickRandomFrom(B(2)⟨stereotype⟩);
18                     if m ∉ M_sample then
19                         selected = TRUE;
20                 if selected then
21                     M_sample.add(m);
22                     Counter⟨stereotype⟩ ++;
```

indexed by the stereotype ($B_{\langle stereotype \rangle}$), and buckets indexed by the number of stereotypes identified in the methods and the stereotypes ($B_{\langle n, stereotype \rangle}$); for instance, $B_{\langle NullVerifier \rangle}$ is the set of methods with the stereotype *NullVerifier*, and the set $B_{\langle 2, Logger \rangle}$ has all the methods with two stereotypes and one of the stereotypes is *Logger*. Note that a method may appear in different buckets $B_{\langle n, stereotype \rangle}$ for a given *n*, because a method can exhibit one or more stereotypes. We also built a second group of buckets indexed by stereotype ($B^{(2)}_{\langle stereotype \rangle}$), but with the methods with LOC in $(Q3, 30]$.

The complete procedure for generating M_{Sample} from the buckets $B_{\langle stereotype \rangle}$, $B^{(2)}_{\langle stereotype \rangle}$, and $B_{\langle n, stereotype \rangle}$ is depicted in Algorithm 1. The first part of the Algorithm (i.e., lines 5 to 10) is to assure that M_{Sample} has a least one method for each combination $\langle n, stereotype \rangle$; then, the second part (i.e., lines 11 to 25) is to balance the selection across different methods exhibiting all the stereotypes. Note that we use a work list to assure sampling without replacement. When we were not able to find methods in $B_{\langle stereotype \rangle}$, we sampled the methods from $B^{(2)}_{\langle stereotype \rangle}$. The charts and values describing M_{Sample} are provided in our online appendix[5].

Regarding the human subjects involved in the study, for the manual identification of stereotypes required for $\mathbf{RQ}_1$, we selected four members of the authors' research lab that did not have any knowledge about the system selection or *TeStereo* internals to avoid bias that could be introduced by the authors, and had multiple years of object-oriented development experience; hereinafter, we will refer to this group of participants as the *Taggers*. For the tasks required with $\mathbf{RQ}_2$ (i.e., writing or evaluating summaries), we contacted (via email) students from the SE classes at the authors' university and external students and researchers. From the participants that accepted the invitation, we selected three groups that we will refer to as $SW_{-TeStereo}$, $SW_{+TeStereo}$, and SR, which stand for summary writers without access to the stereotypes, summary writers with access to the stereotypes, and summary readers, respectively; note that there was no overlap of participants between the three groups. For the evaluation in $\mathbf{RQ}_3$, we mined the list of contributors of the 231 Apache projects; we call this group of participants as *AD* (Apache Developers). We identified the contributors of the projects and contacted them by email to participate in the study. We sent out e-mails listing only the links to the projects to which developers actually contributed (i.e., developers were not contacted multiple times for each project). In the end, we collected 25 completed responses from Apache developers.

4.3 Experimental Design

To answer $\mathbf{RQ}_1$, we randomly split M_{Sample} into two groups, and then we conducted a user study in which we asked four $Taggers$ to manually identify the proposed stereotypes from the methods in both groups (i.e., each $Tagger$ read 105 methods). Before the study, one of the authors met with the $Taggers$ and explained the stereotypes to them. During the study, the methods were displayed to the $Taggers$ in an html-based format using syntax highlighting. After the tagging, we asked the $Taggers$ to review their answers and solve disagreements (if any) after a follow-up meeting. In this meeting, we did not correct the taggers, rather we explained stereotypes that were completely omitted (without presenting the methods from the sample) in order to clarify them; subsequently, the $Taggers$ were able to amend the original tags or keep them the same as they saw fit (we did not urge them to alter any tags). In the end, they provided us with a list of stereotypes for the analyzed methods. We compared the stereotypes identified by *TeStereo* to the stereotypes provided by the $Taggers$. Because of the multi-label classification nature of the process, we measured the accuracy of *TeStereo* by using four metrics widely used with multi-class/label problems [57]: micro-averaging recall (μRC), micro-averaging precision (μPC), macro-averaging recall (MRC), and macro-averaging precision (MPC). The rationale for using micro and macro versions of precision and recall was to measure the accuracy globally (i.e., micro) and at stereotype level (i.e., macro). We discuss the results of $\mathbf{RQ}_1$ in Section 5.1.

To answer $\mathbf{RQ}_2$, for each method in M_{Sample}, we automatically built two html versions (with syntax highlighting) of the source code: with and without stereotype tags. The version with tags was assigned to participants in group $SW_{+TeStereo}$, and the version without tags was assigned to participants in $SW_{-TeStereo}$. Each group of participants had 14 people; therefore, each participant was asked to (i) read 15 methods randomly selected (without replacement) from M_{Sample}, and (ii) write a summary for each method. Note that the participants in $SW_{-TeStereo}$ had no prior knowledge of our proposed stereotypes. In the end, we obtained two summaries for each method of the 210 methods m_i ($14 \times 15 = 210$ methods): one based only on source code ($c^i_{-TeStereo}$), and one based on source code and stereotypes ($c^i_{+TeStereo}$). After collecting the summaries, each of the 14 participants in the group *SR* (i.e., summary readers) were asked to read 15 methods and evaluate the quality of the two summaries written previously for each method. The readers did not know from where the summaries came from, and they got to see the summaries in pairs with the test code at the same time. The quality was evaluated by following a similar procedure and using quality attributes as done in previous studies for automatic generation of documentation [17, 40, 46, 58]. The summaries were evaluated by the participants in terms of completeness, conciseness, and expressiveness. Section 5.2 discusses the results for $\mathbf{RQ}_2$.

Table 2: Accuracy Metrics for Stereotype Detection. The table lists the results for the first round of manual annotation, and second round (in bold) after solving inconsistencies.

Group	μPC	μRC	MPC	MRC
G1	0.87 **(0.98)**	0.82 **(0.89)**	0.82 **(0.99)**	0.77 **(0.92)**
G2	0.80 **(0.95)**	0.89 **(0.94)**	0.80 **(0.94)**	0.84 **(0.94)**

Finally, to answer $\mathbf{RQ}_3$, we distributed a survey to Apache developers in which we asked them to evaluate the usefulness of *TeStereo* reports and stereotypes. The developers were contacted via email; each developer was provided with (i) a *TeStereo* html report that was generated for one Apache project to which the developer contributes, and (ii) a link to the survey. For developers who contributed to multiple Apache projects, we randomly assigned one report (from the contributions). The survey consisted of two parts of questions: background and questions related to *TeStereo* reports and the stereotypes. Section 5.3 lists the questions in the second part; the demographic questions are listed in our online appendix[5]. The answers were analyzed using descriptive statistics for the single/multiple choice questions; and, in the case of open questions, the authors manually analyzed the free text responses using open coding [31]. More specifically, we analyzed the collected data based on the distributions of choices and also checked the free-text responses in depth to understand the rationale behind the choices. The results for $\mathbf{RQ}_3$ are discussed in Section 5.3.

5 EMPIRICAL RESULTS

In this section, we discuss the results for each research question.

5.1 What is TeStereo's accuracy for identifying stereotypes?

Four annotators manually identified stereotypes from 210 unit methods in M_{sample}. Note that the annotators worked independently in two groups, and each group worked with 105 methods. The accuracy of *TeStereo* measured against the set of stereotypes reported by the annotators is listed in Table 2. In summary, there was a total of 102 (2.31%) false negatives (i.e., *TeStereo* missed the stereotype) and 118 (2.68%) false positives (i.e., the $Taggers$ missed the stereotype) in both groups.

We manually checked the false negatives and false positives in order to understand why *TeStereo* failed to identify a stereotype or misidentified a stereotype. *TeStereo* did not detect some stereotypes (i.e., false negatives) in which the purpose is defined by inter-procedural calls, in particular *Logger*, *APIUtilityVerifier* and *InternalCallVerifier*. For instance, the stereotype *Logger* is for unit tests methods performing logging operations by calling the Java `PrintStream` and `Logger` APIs; however, there are cases in which the test cases invoke custom logging methods or loggers from other APIs (e.g., `XmlLogger` from Apache ant). The unit test case in Figure 6 illustrates the issue; while it was tagged as a `Logger` by the $Taggers$, it was not tagged by *TeStereo* because `XmlLogger` is different than the standard Java logging. Few cases of the false negatives were implementation issues; therefore, we used the false positives to improve the stereotypes detection.

Because the $Taggers$ were not able to properly detect some stereotypes (i.e., false positives), we re-explained to them the missed

```
@Test public void test() throws Throwable {
        final XmlLogger logger = new XmlLogger();
        final Cvs task = new Cvs();
        final BuildEvent event = new BuildEvent(task);
        logger.buildStarted(event);
        logger.buildFinished(event);}
```

Figure 6: *Logger* missed by TeStereo.

stereotypes (using the name and rules and without showing methods from the sample); in some cases, participants did not tag methods with the "`Test Initializer`" stereotype, because they did not notice the custom annotation `@Before`. Afterward, we generated a new version of the sample (same methods but with improved stereotypes detection), and then we asked the *Taggers* to perform a second round of tagging. We only asked the annotators to re-tag the methods in the false positive and false negative sets. Finally, we recomputed the metrics, and the results for the second round are shown in bold in Table 2. The results from the second round showed that TeStereo's accuracy improved and the inconsistencies were reduced to 64 (1.45%) false negatives and 25 (0.57%) false positives. The future work will be devoted to improving the data flow analysis and fixing the false negatives.

Summary for RQ$_1$. TeStereo is able to detect stereotypes with high accuracy (precision and recall), even detecting cases in which human annotators fail. However, it has some limitations due to the current implementation of the data-flow based analysis.

5.2 Do the proposed stereotypes improve comprehension of tests cases (i.e., methods in test units)?

To identify whether the stereotypes improve comprehension of methods in unit tests, we measured how good the manually written summaries are when the test cases include (or not) the TeStereo stereotypes. We first collected manually generated summaries from the two participant groups $SW_{+TeStereo}$ and $SW_{-TeStereo}$ as described in Section 4. Then, the summaries were evaluated by a different group of participants who read and evaluated the summaries.

During the "writing" phase we asked the participants to indicate with "N/A" when they were not able to write a summary because of lack of either context or information or they were not able to understand the method under analysis. In 78 out of 420 cases, we got "N/A" as a response from the summary writers; 55 cases were from the participants using only the source code and 23 cases were from participants using the source code and the stereotypes. In total, 64 methods had only one version of the summary available (7 methods had two "N/A"); therefore, the summary readers only evaluated the summaries for 139 (210−64−7) methods in which both versions of the summary were available. Consequently, during the reading phase, 278 summaries were evaluated by 14 participants. It is worth noting that according to the design of the experiment each participant had to evaluate the summaries for 15 methods; however, because of the discarded methods, some of the participants were assigned with fewer than 15 methods. The results for *completeness*, *conciseness*, and *expressiveness* are summarized in Table 3.

Completeness. This attribute is intended to measure whether the summary writers were able to include important information in the summary, which represents a high level of understanding of the code under analysis [46]. In terms of completeness, there is a clear

Table 3: Questions used for RQ$_2$ and the # of answers provided by the participants for the summaries written without ($SW_{-TeStereo}$) and with ($SW_{+TeStereo}$) access to stereotypes.

Do you think the message is complete?	$SW_{-TeStereo}$	$SW_{+TeStereo}$
• Does not miss any imp. info.	46(33.1%)	80(57.6%)
• Misses some important info.	63(45.3%)	47(33.8%)
• Misses the majority of imp. info.	30(21.6%)	12(8.6%)
Do you think the message is concise?	$SW_{-TeStereo}$	$SW_{+TeStereo}$
• Contains no redundant info.	95(68.3%)	95(68.3%)
• Contains some redundant info.	35(25.1%)	35(25.1%)
• Contains a lot of redundant info.	9(6.4%)	9(6.4%)
Do you think the description is expressive?	$SW_{-TeStereo}$	$SW_{+TeStereo}$
• Is easy to read and understand	90(64.7%)	78(56.1%)
• Is somewhat readable	35(25.2%)	42(30.2%)
• Is hard to read and understand	14(10.1%)	19(13.7%)

difference between the summaries written by participants that had the TeStereo stereotypes and those that did not have stereotypes; while 80 summaries from $SW_{+TeStereo}$ were ranked as not missing any information, 46 from $SW_{-TeStereo}$ were ranked in the same category. On the other side of the scale, only 12 summaries from $SW_{+TeStereo}$ were considered to miss the majority of the important info, compared to 30 summaries from $SW_{-TeStereo}$. Thus, the writers assisted with TeStereo stereotypes were able to provide better summaries (in terms of completeness), which suggests that the stereotypes helped them to comprehend the test cases better. Something interesting to highlight here is the fact that some of the writers (from $SW_{+TeStereo}$) included in their summary information based on the stereotypes: "*This is a test initializer.*", "*initialize an empty test case*", "*This method checks whether 'slingId' is null and 'equals' equals to expected.*", "*This is an empty test that does nothing.*" , "*This is an ignored test method which validates if the fixture is installed.*", and "*this setup will be run before the unit test is run and it may throw exception*".

Conciseness. This attribute evaluates if the summaries contain redundant information. Surprisingly, the results are the same for both types of summaries (Table 3); 95 summaries from each group ($SW_{+TeStereo}$ and $SW_{-TeStereo}$) were evaluated as not containing redundant information, and only nine summaries from each group were ranked as including significant amount of redundant information. This is surprising coincidence for which we can not have a clear explanation. However, examples of summaries ranked with a low conciseness show the usage of extra but unrelated information added by the writer: "*Not sure what is going on here, but the end results is checking if r7 == 'ABB: Hello A from BBB'.*", "*Maybe it's testing to see if a certain language is comparable to another, but I can't tell*", and "*this one has an ignore annotation will run like a normal method which is to test the serialize and deserialize performance by timing it.*".

Expressiveness. This attribute aims at evaluating whether the summaries are easy to read. 90 summaries written without having access to the stereotypes were considered as easy to read compared to 78 summaries from the writers with access to the stereotypes. However, when considering the answers for the summaries ranked as easy-to-read or somewhat-readable, both $SW_{+TeStereo}$ and $SW_{-TeStereo}$ account for 86%-90% of the summaries, which are very close. One possible explanation for the slight difference in favor of $SW_{-TeStereo}$ might be that the extra TeStereo tag information could increase the complexity of the summaries. For example, the summary "*This is an 'ignored' test which also does nothing so it makes sure that the program can handle nothing w/o blowing up (it* *throws an exception not just the stack trace).*" is hard to read although it contains the keyword "ignore". Another example is "*setup the current object by assigning values to the tomcat, context, and loader fields.*"

Rationale. We also analyzed the free-text answers provided by the summary readers when supporting their preferences for summaries from $SW_{-TeStereo}$ or $SW_{+TeStereo}$. Overall, 72 explanations claimed that the choice was based on the *completeness* of the summary. Examples include: '*The summary allows for a deeper understanding of what the program is doing and what it is using to make itself work*", "*I prefer this summary because it is more detailed than the other.*", and "*I like this one because it gives you enough information without going overboard*". 52 out of the 72 explanations were for answers in favor of summaries from $SW_{+TeStereo}$. Thus, the rationale provided by the readers reinforces our findings that *TeStereo* helped developers to comprehend the test cases and write better test summaries that include important info.

26 explanations mentioned the *expressiveness* as the main attribute for making their choice: "*This summary is very easy for programmers to understand.*" and "*Easier to read, while I can hardly understand what Summary1 is trying to say.*". In this case, 12 explanations are for readers in favor of summaries from $SW_{+TeStereo}$. Finally, 4 decisions were made based on the *conciseness* of the summaries: "*Slightly more concise*", "*Concise*", "*This concisely explains what is going on with no extra material but it could use a little more information.*", and "*Too much extra stuff in Summary 1*".

Summary for RQ$_2$. The evaluation of the scenario of writing and reading summaries for unit test methods suggests that the proposed unit test stereotypes improve the comprehension of tests cases. The results showed that manually written summaries with assistance from *TeStereo* tags covered more important information than the summaries written without it. In addition, by comparing the evaluation between summaries with and without using *TeStereo* tags, the results indicated that *TeStereo* tags did not introduce redundant information or make the summaries hard to read.

5.3 What are the developers perspectives of the *TeStereo*-based reports for systems in which they contributed?

We received completed surveys from 25 developers of the Apache projects. While the number of participants is not very high, participation is an inherent uncontrollable difficulty when conducting a user study with open source developers. In terms of the highest academic degree obtained by participants, we had the following distribution: one with a high school degree (4%), seven with a Bachelor's degree (28%), sixteen with a Master's degree (64%), and one with Ph.D. (4%). Concerning the programming experience, the mean value is 20.8 years of experience and the median value is 20 years. More specifically, participants had on average 12.9 years of industrial/open-source experience (the median was 14 years). The questions related to **RQ**$_3$ and the answers provided by the practitioners are as the following:

SQ$_1$. Which of the following tasks do you think the tags are useful for? (Multiple-choice and Optional). 48% selected "*Test case comprehension/understanding*", 44 % selected "*Generating summary of unit test case*", 40% vote for the option "*Unit test case maintenance*", and only 8% checked the option "*Debugging unit test cases*".

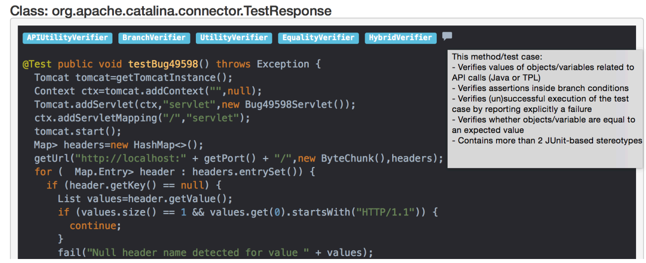

Figure 7: TeStereo documentation for a test case in the Tomcat project.

SQ$_2$. Which of the following tasks do you think the reports are useful for? (Multiple-choice and Optional). 60% selected "*Test case comprehension/understanding*", 48 % selected "*Generating summary of unit test case*", 40% vote for the option "*Unit test case maintenance*", and only 8% checked the option "*Debugging unit test cases*".

SQ$_3$. What tasks(s) do you think the tags/report might be useful for? (Open question) To complement the first two **SQ**s, **SQ**$_3$ aims at examining if the stereotypes and reports are useful from a practitioner's perspective for other software-related tasks. We categorized the responses into the following groups:

- **Unit test quality evaluation**: The participants mentioned the following uses like "*evaluate the quality of the unit tests*", "*a rough categorization [of unit tests] by runtime, e.g. 'fast' and 'slow'*", and "*quality/complexity metrics*".
- **Bad test detection**: two participants suggested that the technique could be used for detecting bad tests. The responses include "*Fixing a system with a lot of bad tests*" & "*probably verifying if there's good 'failure' message*".
- **Code navigation**: One response suggested that the *TeStereo* report is "*a good way to jump into the source code*". This response demonstrates that users can comprehend the test code easier by looking at the *TeStereo* report.

SQ$_4$. Is the summary displayed when hovering over the gray balloon icon useful for you? (Binary-choice). *TeStereo*'s reports include a speech balloon (Figure 7) icon that displays a summary automatically generated by aggregating the descriptions of the stereotypes[2]. We wanted to evaluate usefulness of this feature, and we obtained 14 positive and 11 negative responses. The positive answers were augmented with rationale such as '*It gives the purpose of unit test case glimpsly*", "*Was hard to find, but yes, this makes it easier to grok what you're looking at*", and "*It is clear*". As for the negative answers, the rationale described compatibility issues with mobile devices ("*I am viewing this on an iPad. I can't hover*", "*hovers don't seem to work*"). Yet, some participants found the summary redundant since the info was in the tags.

SQ$_5$. What are the elements that you like the most in the report? (Multiple-choice). Most of the practitioners selected source code box (14 answers, 56%) and test case tags (11 answers, 44%). This suggests that the surveyed practitioners recognize the benefit of the stereotype tags, and are more likely to use the combination

[2]Note that TeStereo's reports (including the balloon and summary features) were only available for the Apache developers.

```
@Test public void testSingleElementRange(){
  final int start=1;
  final int max=1;
  final int step=-1;
  final List seq=new ArrayList();
  final IntegerSequence.Range
        r=IntegerSequence.range(start,max,step);
  for ( Integer i : r) {seq.add(i);}
  Assert.assertEquals(1,seq.size());
  Assert.assertEquals(seq.size(),r.size());
  Assert.assertEquals(start,seq.get(0).intValue());}
```

Figure 8: *InternalCallVerifier* missed by *TeStereo*.

of tags and source code boxes. We received 5 answers (20%) for "*gray balloon icon & summary*", 3 (12%) for "*navigation box*", and 4 (16%) for "*filter*".

SQ_6. Please provide an example of the method that you think the tags are especially useful for unit test case comprehension (Open question). For SQ_6, we collected nine responses in total, and this is related to the open question nature in which some participants filled blank spaces or other characters. One participant mentioned the `testForkModeAlways` method in project `maven` and explained his choice with the following rationale: "*This method is tagged 'BranchVerifier', and arguably it cyclomatic complexity is to great for a test.*" This explanation shows that the stereotype tags (i.e, *BranchVerifier* and *IterativeVerifier*) help developers identify test code that should not include branches/loops. Another response mentions the `testLogIDGenerationWithLowestID` method in project `Ace`; the method was tagged as *Logger* by *TeStereo* and the practitioner augmented his answer with the following: "*Logging in unit tests is usually a code smell, just by looking at this method I realize what* `event.toRepresentation()` *returns is not compared with an expected value.*" This example shows that stereotype tags are also useful for other software maintenance tasks such as code smell detection. Another example is the method `testLogfilePlacement` in `Ant`, and the developer claimed that this is a very good example because the tags helped him to identify that the test case is an internal call verifier. Some responses did not provide the signature of the method, but their comments are useful (e.g., "*TestCleaner is useful to show complexity (hopefully unneeded) of test cases*" and "*I believe they would be useful to check if developers are only developing shallow test cases*").

SQ_7. Please provide an example of the method that you think the tags are NOT useful for unit test case comprehension (open question). For SQ_7, we collected nine valid responses. One example highlights the need for improving the limitations mentioned in Section 5.1, in particular the method `nonExistentHost()` with the following comment from a developer: "*it's about 'verifies (un)successful execution', but it's about expected exception in particular case.*" This issue is due to the fact that *TeStereo* performs over-approximation during static analysis. Although *TeStereo* does not track any branch conditions in the method (some paths may not be executed with a certain input), the over-approximate approach guarantees that potential paths are not missed when *TeStereo* tags the unit test case.

SQ_8. What are the elements that you think need improvement in the report? (Open question). For SQ_8, we collected 13 valid responses. Some practitioners suggested augmenting the reports with summaries describing the method, for example: "*If it can explain about the function, it would be great*" and "*It'd be nicer if the tags conveyed more semantics concepts, rather than mere syntactic properties*". Also, some comments asked for improvement of the user experience in the reports: '*the report should highlight in red the test methods which do contain any assertions*" and "*Being able to collapse the code blocks to make it easier to see summaries*".

SQ_9. What additional information would you find helpful if it were included in the reports? (Open question). These are some sample answers:

- **Test suite quality**: some participants suggested that we need to create a new stereotype to identify redundant test cases, include test coverage info, show evolution of the tags per commits, and indicate size of the method which can be an indicator of methods that need refactoring;
- **Integration**: some practitioners also suggested that we add a link to the full source code on GitHub, so that the code can be seen in its larger context. They also suggested that we integrate *TeStereo* into SonarQube;
- **Detailed description**: these suggestions are more related to personal preferences; for example, "*highlighting the aspect in the code*", "*I would be interested in being able to find which tests check which accessors or methods and vice versa.*", and "*specify what is verified by this method*". The last comment is aligned with the purpose of other summarization approaches such as TestDescriber [52] and UnitTestScribe [40], which generate natural language descriptions of the assertions and focal methods.

Summary for RQ_3. Overall, we obtained 25 responses from active Apache developers, who provided us with useful feedback for improving the stereotypes and the reports. Concerning the usefulness of the stereotypes and the reports, most of the surveyed developers believed that *TeStereo* 's tags and reports are useful for test case comprehension tasks. Other tasks reported by the developers, in which the tags and the reports might be useful, are code smell detection and source code navigation.

5.4 Threats to Validity

Threats to internal validity relate to response bias by participants that either had more difficulty or did not have problems while understanding unit test cases or writing summaries. Based on the results of the study and the large number of the participants, we observed that responses were not dominantly distributed to extremes, which would indicate that these developers were particularly biased based on such difficulty.

The external threats to validity relate to generalizing the conclusions from the study. In our study, we state that these results are based on our sample of unit test cases and participants, but do not claim that these results generalize to all developing systems in other languages and other developers. However, we do present the sampling procedure of unit tests from the whole set of unit test in the 231 Apache projects, which aims to minimize the threat. The selected methods are highly diverse in terms of methods size, method stereotypes, and the number of stereotypes. In addition, we present demographic information of the participants that suggests that we have a diverse sample of developers.

Another threat to validity is that *TeStereo* has some limitations due to the current implementation of the data-flow based analysis. For example, *TeStereo* cannot interpret the variable assignment relations since those require inter-procedural analysis, which leads to false negatives. For example, the unit test case in Figure 8 was annotated as `InternalCallVerifier` by the taggers since the method has a slicing path from variable `r` to `seq`, and `IntegerSequence.range(start,max,step)` at line 6 is an internal method call. However, *TeStereo* cannot interpret the variable assignment relations in the for-loop (line 7), since it needs to understand the assignment relations in "*Integer i:r*" and "*seq.add(i)*". Due to this limitation, *TeStereo* loses the backward tracking to the internal function call.

6 RELATED WORK

There are some related techniques for studying unit test cases, which include unit test case minimization [37, 38], prioritization [20, 56, 59], test case descriptions [36, 40, 52, 68], code quality [10], test coverage [33], data generation [39, 43], unit test smells [14, 44, 61–63], fault localization [65], automatic test case generation [18, 25, 27, 37, 60, 64, 66], and automatic recommendation of test examples [53]. *TeStereo* is also related to (i) techniques for generating documentation for software artifacts [23, 34, 41, 42, 47], and (ii) other approaches for supporting code comprehension provided by difference tools [26, 40, 46, 49, 52]. Compared to the existing approaches, *TeStereo* is novel in that it considers stereotypes at the test suite level.

6.1 Stereotypes Definition and Detection

Several studies [21, 23, 24] focused on classifying software entities, such as methods, classes, and repository commits. Generally, the studies classify software entities as different stereotypes based on static analysis techniques and predefined rules [23, 24]. Dragan et al. first presented and defined taxonomy of method stereotypes [23]. The authors implemented a tool, namely `StereoCode`, which automatically identifies method stereotypes for all methods in a system. Later, Dragan et al. extended the classification of stereotypes to class level granularity by considering frequency and composition of the method stereotypes in one class [24]. The results showed that 95% of the classes were stereotyped by their approach. Dragan et al. further refined stereotypes at the commit level [21]. The categorization of a commit is based on the stereotype of the methods that are added/deleted in the commit. Different from Dragan et al.'s implementation that works on C++, Moreno and Marcus [48] implemented a classification tool, named `JStereoCode`, for automatically identifying method and class stereotypes in Java systems. Andras et al. measured runtime behavior of methods and method calls to reflect method stereotypes [9]. Their observation showed that most methods behave as expected based on the stereotypes. Overall, none of the existing studies focus on stereotype classification of unit test cases. *Our approach is the first one to define and classify unit test case stereotypes by (i) analyzing unit test API calls and (ii) performing static analysis on data/control flows.*

6.2 Utilizing Stereotypes for Automatic Documentation

A group of approaches and studies utilize stereotype identification for other goals. Linares-Vásquez et al. [17, 41] implemented a tool, namely `ChangeScribe`, for automatically generating commit messages. `ChangeScribe` extracts changes between two adjacent versions of a project and identifies involved change types in addition to performing commit level stereotype analysis. Dragan et al. showed that the distribution of method stereotypes could be an indicator of system architecture/design [22]. In addition, their technique could be utilized in clustering systems with similar architecture/design. Moreno et al. [46, 49] and Abid et al. [7] utilized class stereotypes to summarize the responsibilities of classes in different programming languages (Java and C++) respectively. Ghafari et al. [30] used stereotypes to detect focal methods (methods responsible for system state changes examined through assertions in unit tests) in a unit test case. *Overall, our work is first to improve unit test comprehension and test suite navigation by using unit test stereotypes.*

6.3 Automatic Documentation of Unit Test Cases

Kamimura and Murphy presented an approach for automatically summarizing JUnit test cases [36]. Their approach identified the focal method based on the number of invocations of the method. Panichella et al. [52] presented an approach, `TestDescriber`, for generating test case summaries on automatically generated JUnit test cases. The summary contains three different levels of granularity: class, method, and test level (i.e., branch coverage). Furthermore, Li et al. [40] proposed `UnitTestScribe` that combines static analysis, natural language processing, and backward slicing techniques to automatically generate detailed method-level summarization for unit test cases. Zhang et al. [67, 68] presented a natural language-based approach that extracts the descriptive nature of test names to generate test templates. *Overall, none of the existing techniques for documenting unit test cases focuses on unit test case stereotypes, besides our approach.*

7 CONCLUSION

In this paper, we first presented a novel catalog of stereotypes for methods in unit tests to categorize JUnit test cases into 21 stereotypes; the catalog aims at improving program comprehension of unit test, when the unit test methods are annotated with the stereotypes. We propose an approach, *TeStereo*, for automatically tagging stereotypes for unit tests by performing control-flow, data-flow, and API call based static analyses on the source code of a unit test suite. *TeStereo* also generates html reports that include the stereotype tags, source code, and navigation features to improve the comprehension and browsing of unit tests in a large test suite.

To validate *TeStereo*, we conducted empirical studies based on 231 Apache projects, 46 students and researchers and a survey with 25 Apache developers. Also, we evaluated 420 manually generated summaries and 210 unit test methods with and without stereotype annotations. Our results show that (i) *TeStereo* achieves very high precision and recall (0.99 & 0.94) in terms of annotating unit test stereotypes, (ii) the proposed stereotypes improve comprehension of unit test cases in software maintenance tasks, and (iii) most of the developers agreed that *TeStereo* stereotypes and reports are useful. Our results demonstrate that TeStereo's tags are useful for test case comprehension tasks as well as other tasks, such as code smell detection and source code navigation.

8 ACKNOWLEDGEMENTS

We would like to thank the Apache developers that participated in the survey and provided meaningful feedback. Additionally, we would like to thank the individuals that participated in our study.

REFERENCES

[1] *JMockit.* http://jmockit.org/.
[2] *JUnit.* http://junit.org/.
[3] *Mockito.* http://mockito.org/.
[4] *Nunit.* http://www.nunit.org/.
[5] *Online appendix.* https://sites.google.com/site/testereoonline/.
[6] *PHPUnit.* https://phpunit.de/.
[7] Nahla J. Abid, Natalia Dragan, Michael L. Collard, and Jonathan I. Maletic. 2015. Using Stereotypes in the Automatic Generation of Natural Language Summaries for C++ Methods. In *Proc. ICSME*. IEEE, 561–565.
[8] Nouh Alhindawi, Natalia Dragan, Michael L. Collard, and Jonathan I. Maletic. 2013. Improving Feature Location by Enhancing Source Code with Stereotypes. DOI:http://dx.doi.org/10.1109/ICSM.2013.41
[9] Peter Andras, Anjan Pakhira, Laura Moreno, and Andrian Marcus. A measure to assess the behavior of method stereotypes in object-oriented software. In *WETSoM 2013*. IEEE.
[10] Mauricio F Aniche, Gustavo A Oliva, and Marco A Gerosa. 2013. What do the asserts in a unit test tell us about code quality? a study on open source and industrial projects. In *CSMR'13*. IEEE, 111–120.
[11] David Astels. 2003. *Test-Driven Development: A Practical Guide: A Practical Guide.* Prentice Hall PTR.
[12] Luciano Baresi and Matteo Miraz. 2010. TestFul: Automatic Unit-test Generation for Java Classes. In *Proceedings of ICSE10*. ACM, New York, NY, USA, 281–284. DOI:http://dx.doi.org/10.1145/1810295.1810353
[13] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. 2013. The evolution of project inter-dependencies in a software ecosystem: The case of apache. In *ICSM*. IEEE, 280–289.
[14] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and David Binkley. 2012. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *ICSM'12 28th*. IEEE, 56–65.
[15] Kent Beck. 2003. *Test Driven Development: By Example.* Addison-Wesley Professional.
[16] Kent Beck and Cynthia Andres. 2004. *Extreme Programming Explained: Embrace Change* (2nd ed.). Addison-Wesley.
[17] Luis Fernando Cortés-Coy, Mario Linares-Vásquez, Jairo Aponte, and Denys Poshyvanyk. 2014. On automatically generating commit messages via summarization of source code changes. In *SCAM'14*. IEEE, 275–284.
[18] Ermira Daka, José Campos, Gordon Fraser, Jonathan Dorn, and Westley Weimer. 2015. Modeling readability to improve unit tests. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM.
[19] Ermira Daka and Gordon Fraser. 2014. A Survey on Unit Testing Practices and Problems. In *ISSRE'14*. 201–211. DOI:http://dx.doi.org/10.1109/ISSRE.2014.11
[20] Daniel Di Nardo, Nadia Alshahwan, Lionel Briand, and Yvan Labiche. 2013. Coverage-based test case prioritisation: An industrial case study. In *ICST'13*. IEEE, 302–311.
[21] Natalia Dragan, Michael L Collard, Maen Hammad, and Jonathan I Maletic. 2011. Using stereotypes to help characterize commits. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*. IEEE, 520–523.
[22] Natalia Dragan, Michael L Collard, and Jonathan Maletic. Using method stereotype distribution as a signature descriptor for software systems. In *ICSM 2009*. 567–570.
[23] Natalia Dragan, Michael L Collard, and Jonathan I Maletic. 2006. Reverse engineering method stereotypes. In *Software Maintenance, 2006. ICSM'06. 22nd IEEE International Conference on*. IEEE, 24–34.
[24] Natalia Dragan, Michael L Collard, and Jonathan I Maletic. 2010. Automatic identification of class stereotypes. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*. IEEE, 1–10.
[25] Faezeh Ensan, Ebrahim Bagheri, and Dragan Gašević. 2012. Evolutionary search-based test generation for software product line feature models. In *AISE*. Springer, 613–628.
[26] J. Fowkes, P. Chanthirasegaran, R. Ranca, M. Allamanis, M. Lapata, and C. Sutton. 2017. *IEEE Transactions on Software Engineering* (2017).
[27] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *FSE'11*. ACM, 416–419.
[28] Gordon Fraser and Andrea Arcuri. 2013. Whole Test Suite Generation. *IEEE Trans. Softw. Eng.* 39, 2 (Feb. 2013), 276–291. DOI:http://dx.doi.org/10.1109/TSE.2012.14
[29] Juan Pablo Galeotti, Gordon Fraser, and Andrea Arcuri. 2014. Extending a Search-based Test Generator with Adaptive Dynamic Symbolic Execution. In *Proceedings of the 2014 ISSTA (ISSTA 2014)*. ACM, 421–424. DOI:http://dx.doi.org/10.1145/2610384.2628049
[30] Mohammad Ghafari, Carlo Ghezzi, and Konstantin Rubinov. 2015. Automatically identifying focal methods under test in unit test cases. In *SCAM'15*. IEEE, 61–70.
[31] Barney G. Glaser and Anselm L. Strauss. 1967. *The Discovery of Grounded Theory: Strategies for Qualitative Research.* Aldine de Gruyter.
[32] Andy Hunt and Dave Thomas. 2003. *Pragmatic Unit Testing in Java with JUnit.* The Pragmatic Programmers.
[33] Chen Huo and James Clause. 2016. Interpreting Coverage Information Using Direct and Indirect Coverage. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 234–243.
[34] Daniel Jackson and David A Ladd. 1994. Semantic Diff: A Tool for Summarizing the Effects of Modifications.. In *ICSM*, Vol. 94. 243–252.
[35] Ranjit Jhala and Rupak Majumdar. 2005. Path slicing. In *ACM SIGPLAN Notices*, Vol. 40. ACM, 38–47.
[36] Manabu Kamimura and Gail C Murphy. 2013. Towards generating human-oriented summaries of unit test cases. In *ICPC'13*. IEEE, 215–218.
[37] Yong Lei and James H Andrews. 2005. Minimization of randomized unit test cases. In *Software Reliability Engineering, 2005. ISSRE 2005. 16th IEEE International Symposium on*. IEEE, 10–pp.
[38] Andreas Leitner, Manuel Oriol, Andreas Zeller, Ilinca Ciupa, and Bertrand Meyer. 2007. Efficient unit test case minimization. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 417–420.
[39] Boyang Li, Mark Grechanik, and Denys Poshyvanyk. 2014. Sanitizing and minimizing databases for software application test outsourcing. In *ICST 2014*. IEEE, 233–242.
[40] B. Li, C. Vendome, M. Linares-Vasquez, D. Poshyvanyk, and N.A. Kraft. 2016. Automatically Documenting Unit Test Cases. In *Proceedings of ICST'16*.
[41] Mario Linares-Vásquez, Luis Fernando Cortés-Coy, Jairo Aponte, and Denys Poshyvanyk. 2015. Changescribe: A tool for automatically generating commit messages. In *37th IEEE/ACM ICSE'15, Formal Research Tool Demonstration*.
[42] M. Linares-Vasquez, B. Li, C. Vendome, and D. Poshyvanyk. Documenting Database Usages and Schema Constraints in Database-Centric Applications. In *ISSTA'16*.
[43] Ruchika Malhotra and Mohit Garg. 2011. An adequacy based test data generation technique using genetic algorithms. *Journal of information processing systems* 7, 2 (2011), 363–384.
[44] Gerard Meszaros. 2007. *xUnit test patterns: Refactoring test code.* Pearson Education.
[45] Audris Mockus, Roy T Fielding, and James D Herbsleb. 2002. Two case studies of open source software development: Apache and Mozilla. *ACM TOSEM* 11, 3 (2002), 309–346.
[46] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock, and K Vijay-Shanker. Automatic generation of natural language summaries for java classes. In *ICPC 2013*. IEEE, 23–32.
[47] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrian Marcus, and Gerardo Canfora. 2014. Automatic generation of release notes. In *ESEC/FSE'14*. ACM, 484–495.
[48] Laura Moreno and Andrian Marcus. 2012. Jstereocode: automatically identifying method and class stereotypes in java code. In *ASE'12*. ACM, 358–361.
[49] Laura Moreno, Andrian Marcus, Lori Pollock, and K Vijay-Shanker. 2013. Jsummarizer: An automatic generator of natural language summaries for java classes. In *ICPC 2013*. IEEE, 230–232.
[50] D. North. *Introducing BDD. http://dannorth.net/introducing-bdd*.
[51] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation *(ICSE '07)*. IEEE Computer Society, 10. DOI:http://dx.doi.org/10.1109/ICSE.2007.37
[52] Sebastiano Panichella, Annibale PanichellaâĂŇ, Moritz BellerâĂŇ, Andy Zaidman, and Harald Gall. 2016. The impact of test case summaries on bug fixing performance: An empirical investigation. In *ICSE'16*.
[53] Raphael Pham, Yauheni Stoliar, and Kurt Schneider. Automatically recommending test code examples to inexperienced developers. In *FSE'15*. ACM, 890–893.
[54] Peter C Rigby, Daniel M German, and Margaret-Anne Storey. 2008. Open source software peer review practices: a case study of the apache server. In *Proceedings of the 30th ICSE*. ACM, 541–550.
[55] José Miguel Rojas, José Campos, Mattia Vivanti, Gordon Fraser, and Andrea Arcuri. 2015. Combining multiple coverage criteria in search-based unit test generation. In *International Symposium on Search Based Software Engineering*. Springer, 93–108.
[56] Gregg Rothermel, Roland H Untch, Chengyun Chu, and Mary Jean Harrold. 2001. Prioritizing test cases for regression testing. *TSE* 27, 10 (2001), 929–948.
[57] Marina Sokolova and Guy Lapalme. 2009. A systematic analysis of performance measures for classification tasks. *Information Processing & Management* 45, 4 (2009), 427–437.
[58] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. 2010. Towards automatically generating summary comments for java methods. In *ASE*. ACM, 43–52.
[59] Panagiotis Stratis and Ajitha Rajan. 2016. Test Case Permutation to Improve Execution Time. In *31th IEEE/ACM International Conference on Automated Software Engineering (ASE)*.
[60] Hongyin Tang, Guoquan Wu, Jun Wei, and Hua Zhong. 2016. Generating test cases to expose concurrency bugs in android applications. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 648–653.
[61] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Penta, Rocco Oliveto, Andrea Lucia, and Denys Poshyvanyk. 2016. An Empirical Investigation into the Nature of Test Smells. In *ASE'16*.
[62] Arie Van Deursen and Leon Moonen. 2002. The video store revisited–thoughts on refactoring and testing. In *Proc. 3rd Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering*. Citeseer, 71–76.
[63] Arie van Deursen, Leon Moonen, Alex van den Bergh, and Gerard Kok. 2001. *Refactoring test code.* CWI.

[64] Westley Weimer. 2015. Generating Readable Unit Tests for Guava. In *SSBSE 2015, Bergamo, Italy, September 5-7, 2015*, Vol. 9275. Springer, 235.

[65] Jifeng Xuan and Martin Monperrus. 2014. Test case purification for improving fault localization. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 52–63.

[66] Akihisa Yamada, Takashi Kitamura, Cyrille Artho, Eun-Hye Choi, and Armin Biere. 2016. Greedy Combinatorial Test Case Generation Using Unsatisfiable Cores. In *ASE'16*.

[67] Benwen Zhang, Emily Hill, and James Clause. 2015. Automatically Generating Test Templates from Test Names. In *ASE 2015*. IEEE, 506–511.

[68] Benwen Zhang, Emily Hill, and James Clause. 2016. Towards Automatically Generating Descriptive Names for Unit Tests. In *ASE'16*.