Studying Test Annotation Maintenance in the Wild

Dong Jae Kim*, Nikolaos Tsantalis†, Tse-Hsun (Peter) Chen*, Jinqiu Yang†
*Software PEformance, Analysis and Reliability (SPEAR) Lab,
Concordia University, Montreal, Canada
†Department of Computer Science and Software Engineering,
Concordia University, Montreal, Canada
{k_dongja, nikolaos, peterc, jinqiuy}@encs.concordia.ca

***Abstract*—Since the introduction of annotations in Java 5, the majority of testing frameworks, such as JUnit, TestNG, and Mockito, have adopted annotations in their core design. This adoption affected the testing practices in every step of the test life-cycle, from fixture setup and test execution to fixture teardown. Despite the importance of test annotations, most research on test maintenance has mainly focused on test code quality and test assertions. As a result, there is little empirical evidence on the evolution and maintenance of test annotations. To fill this gap, we perform the first fine-grained empirical study on annotation changes. We developed a tool to mine 82,810 commits and detect 23,936 instances of test annotation changes from 12 open-source Java projects. Our main findings are: (1) Test annotation changes are more frequent than rename and type change refactorings. (2) We recover various migration efforts within the same testing framework or between different frameworks by analyzing common annotation replacement patterns. (3) We create a taxonomy by manually inspecting and classifying a sample of 368 test annotation changes and documenting the motivations driving these changes. Finally, we present a list of actionable implications for developers, researchers, and framework designers.**

***Index Terms*—Software Quality, Empirical Study, Annotation, Software Evolution**

I. Introduction

Modern software systems are becoming more complex due to the ever-growing demands from customers. To ensure that the software quality remains on par with consumer expectations, testing has become a pivotal role in software development. Developers rely on testing to verify the quality of every code change and provide an indication on whether the software can be released to production [1].

To increase the effectiveness of testing, developers need to maintain and improve test code continuously. Similar to source code, test code may also contain design issues that hinder the quality. For example, prior studies have found that the results of some test cases can be unreliable (i.e., flaky tests) due to bugs in test code [2], [3]. To that end, developers have begun to notice a recurring design problem in test code and coined the term test smells [4] as an indicator of design problems in tests. Since its inception, researchers have shown that test smells are prevalent in software systems [5], negatively affect software maintainability and comprehension [6], [7], and may impact software quality in terms of post-release defects [8].

The introduction of annotations in Java 5 has driven annotation as a critical component of the many Java-based frameworks, influencing how developers to design and implement software. Even in software testing, frameworks such as JUnit, TestNG, and Mockito have all adopted annotations as critical ingredients in test design and implementation. A prior study [9] has found that JUnit4 is one of the most widely utilized testing frameworks for Java-based systems, and test annotations (e.g., @Test) are also one of the most widely used annotations in Java development. Table I provides an overview of the commonly used test annotations in JUnit4. Although the test annotations may be different across testing frameworks (e.g., TestNG or JUnit5), in general, they provide similar functionalities.

Despite the importance of test annotations, most prior research on test maintenance has only focused on general test design and test assertions [6], [7], [10]–[14] and has not considered the peculiarity of test annotations. Therefore, in this paper, we present the first empirical study on how developers leverage test annotations in the wild to maintain the high quality of test code (e.g., readability, test flakiness, test performance, obsolete test). We first extended the state-of-the-art refactoring mining tool, RefactoringMiner 2.0 [15], to detect annotation additions, removals, and modifications. We study the collected annotation changes both quantitatively and qualitatively by answering three research questions:

RQ1: How common are test annotation changes? Test annotation changes are 26.5% more common than regular test refactorings such as renames and type changes. Despite their popularity, there is negligible tool support (e.g., antipattern detection, annotation change suggestions, and annotation API usages) for test annotation changes.

RQ2: How are test annotations changed in the wild? We quantitatively study frequent test annotation changes. We find that developers update test annotations more frequently than annotation additions and removals. Moreover, test annotation migration across testing frameworks and within a different version of the same framework is also common.

RQ3: Why do developers change test annotations? We qualitatively uncover test annotation usage and misusage, and how developers bypass the limitations of test annotations. Our findings highlight potential future directions on helping developers improve test maintenance and detect potential issues in test code.

In summary, our findings provide actionable implications for three groups of audiences:

(1) **Researchers**: We open an avenue for further research

TABLE I
A BRIEF OVERVIEW ON COMMONLY USED JUNIT4 ANNOTATIONS.

Annotations	Annotation Location	Description of Commonly used JUnit Annotation
@Rule	Field	@Rule provides a mechanism to enhance tests by running some code around a test case execution, which is similar to fixture and teardown.
@Parameterize	Field/Method	Test case annotated with @Parameterize can be invoked by using a predefined input (i.e., parameterized test inputs) and expected output.
@Test	Method	@Test indicates that the annotated test code should be executed as a test case. @Test takes optional parameters, such as Timeout to indicate that the test should finish within a given time, or exception to indicate that the test should throw an exception.
@Before/@After	Method	@Before indicates that the annotated test code should be executed as a precondition before each test case (i.e., Database setup). Similarly, @After indicates the execution of the annotated test code as a postcondition after each test case.
@BeforeClass/@AfterClass	Method	@BeforeClass and @AfterClass are similar to @Before and @After annotation types, but indicate the annotated test code to only execute once (i.e., before or after the test class is invoked).
@Ignore	Method/Class	@Ignore indicates that the annotated test case should not execute.
@Category	Method/Class	@Category provides a mechanism to label and group tests, giving developers the option to include or exclude groups from execution.
@Test(timeout=X)	Method/Class	A test will fail, if its execution takes longer than the value X specified in timeout.

directions on detecting misuses and test smells related to test annotations and their relation with other aspects of software development (i.e., quality, maintainability, and performance improvements). We also highlight potential directions on automated test code refactoring by leveraging test annotations.

(2) **Developers**: Our findings reveal the usage of test annotations in an ad-hoc manner by some developers. A number of test cases is temporarily disabled until a fix is found; however, the disabled tests are not re-enabled after the fix. Moreover, in several cases, developers are unaware of the features offered by testing frameworks, and thus apply suboptimal custom solutions. These findings indicate the need to educate developers with the best testing practices and provide recommendation tools to help developers apply the appropriate test annotations where needed.

(3) **Framework Designers**: We find cases where developers try to bypass the current limitations of test annotations (i.e., fixture configuration) and provide suggestions for framework designers on improving the flexibility of test annotations.

Paper organization. Section II surveys related work. Section III discusses our extension on RefactoringMiner 2.0 and provides preliminary results. Section IV presents our quantitative analysis results, and Section V presents our qualitative analysis results. Section VI summarizes the implication of our findings. Section VII discusses threats to validity. Section VIII concludes the paper.

II. RELATED WORK

In this section, we discuss prior research in the areas of test maintenance and evolution, Java annotation usage, and test framework usage.

Test Maintenance and Evolution. Researchers have been studying testing practice, especially on test quality and evolution. Pinto et al. [16] studied how the test code evolves in practice for more effective automatic program repair. Zaidman et al. [17] studied how the test code co-evolves with production code. Bavota et al. [7] found that test smells are prevalent in software systems and may hinder test comprehension and maintenance. Ma'ayan et al. [18] quantitatively studied the language features of JUnit assertions and identified opportunities for improvement. Researchers further correlated assertions with defects [19] and test suite effectiveness [20]. Our work is the first to study the maintenance of test annotations and document test annotation usages, misuses, and limitations.

Empirical Studies on Java Annotations. Rocha et al. [21] mined 106 open-source Java systems to investigate annotation usage empirically. Similarly, Dyer et al. [22] analyzed 31K open-sourced projects to analyze how various Java language features are adopted by developers, including the adoption of Java annotations. Their study shows that Java annotations are very commonly adopted. Parnin et al. [23] used 40 open-source Java projects to study the adoption of Java generics and contrast it with the adoption of annotations. This study shows that a champion leads annotation adoption in a development team. Yu et al. [24] performed the first large-scale empirical study about Java annotation usage, evolution and impact without focusing on test-related annotations. In contrast to previous studies on Java annotation usage, we perform the first in-depth study on how test annotations from various frameworks are utilized and maintained, and derive the first taxonomy of annotation changes. Moreover, our study yields actionable implications for researchers, developers, and testing framework designers to further expand and improve test annotation practices.

Testing Framework Usage. Researchers have been investigating how developers use various testing and mocking frameworks in open-source systems. Zaraouili and Mens performed one of the first studies on the evolution of the testing frameworks [9]. Their study shows that JUnit was the most prominent testing library, while many libraries are used simultaneously, e.g., PowerMock, Mockito, and EasyMock complement each other. Researchers have conducted quantitative studies to understand the adoption of mocking frameworks in mocking file dependencies [25], and in general use [26]. More recently, Spadini et al. [27] performed a comprehensive study on how and why developers use mocking in test code, and how mocking evolves over time. Different from prior work, we zoom in and perform an in-depth analysis at a more fine-grained level (i.e., annotation changes at commit level) to understand how test annotations are evolved and maintained in practice.

III. METHODOLOGY

RefactoringMiner Extension. In order to detect annotation additions, removals and modifications, we extended the state-of-the-art refactoring mining tool, RefactoringMiner 2.0 [15]. We selected this tool for the following reasons:

1) It operates at commit level, allowing us to obtain annotation changes at the finest granularity level of software evolution (i.e., commits).
2) It can detect refactoring operations, allowing us to include annotation changes for refactored program elements (i.e., methods with changes in their signatures, moved/renamed classes and fields) in addition to non-refactored program elements. This makes our dataset more complete and our findings more reliable.
3) It has the highest precision (96.6%) and recall (94%) among other refactoring mining and AST diff tools, allowing us to have an accurate dataset of annotation changes with a very small number of false positives and false negatives.
4) It has the fastest execution time among other refactoring mining tools, allowing us to scale up our data collection for the entire commit history of large projects with over 20K commits.

Using the RefactoringMiner API, we obtain the pairs of program elements (i.e., type, method and field declarations), which have been matched between the currently analyzed commit and its parent in the directed acyclic graph that models the commit history of git-based version control repositories. The pairs of matched program elements may have identical signatures (e.g., a pair of methods with identical names, parameter and return types), or may have different signatures due to refactoring operations (e.g., RENAME METHOD, CHANGE PARAMETER TYPE, ADD/DELETE PARAMETER).

Java annotations are used in three different forms:

- **Marker** annotations without member value pairs: `@TypeName`.
- **Normal** annotations with a list of member value pairs: `@TypeName(name1=value1,name2=value2,..)`, where names are `SimpleName` AST nodes and values are `Expression` AST nodes.
- **Single Member** annotations with a single member value: `@TypeName(Expression)`, where the member name is omitted (i.e., `@foo(bar)` is equivalent to the normal annotation `@foo(name=bar)`).

We consider two annotations as equal if they have the same `TypeName`, and the same member value pairs regardless of their order. Let us assume that for a given pair of matched program elements A_p is the annotation set of the program element in the parent commit, and A_c is the annotation set of the matched program element in the child commit. Then, the added annotations are computed as $A^+ = A_c \setminus (A_p \cap A_c)$. The removed annotations are computed as $A^- = A_p \setminus (A_p \cap A_c)$. The pairs of modified annotations are computed as $A^{\sim} = \{(a_p, a_c) | a_p \in A_p \ \wedge a_c \in A_c \ \wedge a_p.\texttt{TypeName} = a_c.\texttt{TypeName} \ \wedge a_p.\texttt{MemberValuePairs} \neq a_c.\texttt{MemberValuePairs}\}$.

To evaluate the precision and recall of our RefactoringMiner extension, we extended the oracle used in [15], which contains true refactoring instances found in 536 commits from 185 open-source GitHub projects, with instances of Annotation Additions/Removals/Modifications for four different program elements, namely type, method, field, and parameter declarations. To compute precision, an author of the paper manually validated 638 annotation change instances reported by our RefactoringMiner extension. To compute recall, we need to find all true instances of annotation changes. We followed the same approach as in [15] by executing a second tool, namely GumTree [28], and considering as the ground truth the union of the true positives reported by RefactoringMiner and GumTree. GumTree takes as input two abstract syntax trees (e.g., Java compilation units) and produces the shortest possible edit script to convert one tree to another. We used all *Insert* and *Delete* edit operations on Annotation AST nodes to extract annotation changes and report them in the same format used by RefactoringMiner. Table II shows the number of true positives (TP), false positives (FP), and false negatives (FN) detected/missed by our RefactoringMiner extension. The overall precision is 99.7% and the recall is 98.7%.

TABLE II
PRECISION AND RECALL OF OUR EXTENDED VERSION OF REFACTORINGMINER.

Change Type	TP	FP	FN	Precision	Recall
Add Method Annotation	312	1	7	99.7%	97.8%
Remove Method Annotation	97	1	0	99%	100%
Modify Method Annotation	19	0	0	100%	100%
Add Parameter Annotation	29	0	0	100%	100%
Remove Parameter Annotation	3	0	0	100%	100%
Modify Parameter Annotation	2	0	0	100%	100%
Add Field Annotation	47	0	1	100%	97.9%
Remove Field Annotation	17	0	0	100%	100%
Modify Field Annotation	7	0	0	100%	100%
Add Class Annotation	52	0	0	100%	100%
Remove Class Annotation	20	0	0	100%	100%
Modify Class Annotation	31	0	0	100%	100%
Overall	636	2	8	99.7%	98.7%

Studied Systems. We choose the studied systems by following three selection criteria. First, we selected the top 1,000 Java projects on GitHub ordered by popularity (i.e., stargazer count). We also made sure that the repositories are not forks. Second, we discarded projects that are below 90 percentile in terms of size (i.e., lines of code), repository popularity (i.e., stars) and the number of commits. Finally, we discarded inactive repositories that did not have any commits in 2020. We ended up with 12 systems, i.e., Druid, Hadoop, Cassandra, Storm, Flink, Hbase, Camel, Hive, Openfire, Ambari, OrientDB and Kafka. These studied systems cover different domains, ranging from distributed databases, stream processing frameworks, message brokers, and groupchat servers. Table III shows an overview of the studied systems.

Our study focuses on test annotation usage, but there may be some non-test-related annotations in test classes. Hence, we set off to understand what are the common testing frameworks or libraries from which test-related annotations are used. We mined all annotation usages in the versions released in 2015 and 2020, respectively, for the 12 studied systems. In particular, we analyzed all test files that have a *".java* extension and "[Tt]est(s*)" as prefix or suffix in their name. We manually verify the build configuration files (e.g., Maven or Gradle build file) of the studied systems to use the default heuristic

TABLE III
AN OVERVIEW OF THE STUDIED SYSTEMS (FROM 2015 TO 2020).

Systems	Total Test LOC (2015 → 2020)	No. Test Method (2015 → 2020)	No. Test Class (2015 → 2020)
Ambari	125K → 273.8K	2,471 → 5,753	501 → 999
Camel	562K → 787K	12,884 → 18,693	6,713 → 8,961
Cassandra	44.3K → 189K	969 → 4,515	217 → 626
Druid	45K → 307K	773 → 5,818	199 → 1,148
Flink	79K → 437K	1,416 → 9,199	412 → 2,150
Hadoop	480K → 914K	9,269 → 17,610	1,798 → 2,954
Hbase	185K → 359K	2,843 → 5,861	660 → 1,476
Hive	124K → 323K	2,572 → 7,541	473 → 1,182
Ignite	261K → 99K	4,146 → 2,286	1,285 → 529
Kafka	2.9K → 191K	77 → 6,059	24 → 688
Openfire	2.2K → 8.3K	84 → 361	25 → 51
Storm	3.7K → 47K	118 → 1,134	35 → 277
Total	1916K → 3939K	37,622 → 84,830	12,342 → 21,041

TABLE IV
USE OF ANNOTATIONS FROM DIFFERENT FRAMEWORKS IN TEST CODE RELEASED IN 2015 AND 2020, RESPECTIVELY.

Framework Type	Frequency	Proportion (%)
Testing Framework	**32,900 → 102,395**	**72.6% → 65%**
JUnit	31,256 → 101,047	69% → 64%
TestNG	1,644 → 1,348	3.6% → < 1%
Mocking Framework	**260 → 1,640**	**< 1% → 1%**
Mockito	236 → 935	< 1% → < 1%
PowerMock	24 → 256	< 1% → < 1%
EasyMock	0 → 449	0% → < 1%
Java lang annotations	**8,579 → 16,125**	**19% → 10%**
Custom Annotation	**2,738 → 36,300**	**1.7% → 23%**
Spring Framework	**442 → 551**	**1% → < 1%**
Other Libraries E.g., Google, JavaX	**410 → 17,369**	**1% → 1%**

specified by Maven/Gradle plugin to identify test files. After collecting the annotation usages, we manually study them and identify their corresponding framework. Table IV summarizes the annotation usage of different frameworks in the 12 studied systems. We find that JUnit annotations are the most commonly used annotations in test code, accounting for 69% of the mined annotations in 2015 and 64% in 2020. TestNG is a less commonly used testing framework, accounting for 3.6% of the mined annotations in 2015 and 1% in 2020. Our finding shows that developers in the studied systems are migrating away from TestNG. We also found annotations from frameworks used for test mocking (i.e., Mockito, PowerMock and EasyMock), the Spring framework, and other non-test-related libraries. The annotation usage of testing frameworks, such as JUnit and mocking frameworks, increases significantly over the years. However, their percentages decrease due to the increasing use of custom annotations. In section VI, we discuss some of the custom annotations related to testing.

To collect annotation changes, we run our RefactoringMiner extension on every commit that modified at least one Java test file between 2015-2020 for the 12 studied systems. We only keep changes on test-related annotations (i.e., from JUnit, TestNG and mocking frameworks) and discard the rest. In total, we mined 109,460 test-related annotation changes in the commit history of the 12 studied systems from 2015 to 2020.[1]

IV. A QUANTITATIVE STUDY ON TEST ANNOTATIONS

In this section, we conduct a quantitative study to understand the prevalence of test annotation usages and change patterns. In particular, we answer two research questions:

TABLE V
MINED TEST CODE CHANGES IN 82,810 COMMITS.

	Field		Method		Class		Total	Num.
	#	# Per Commit	#	# Per Commit	#	# Per Commit	#	Commits
Annotation Changes	396	0.075	18,472	3.52	5,068	0.97	23,936	5,249
Refactoring	7,125	0.40	9,657	0.53	2,136	0.12	18,918	17,914
Δ % Percentage	-94.4%	-81.3%	+91.3%	+564.2%	+137.2%	+708.3%	+26.5%	-70.7%

RQ1: How common are test annotation changes? As a stepping stone to understanding how developers leverage annotations, we examine how frequently test annotations are changed compared to common source code changes (i.e., renames and type changes) at the same program element level.
RQ2: How are test annotations changed in the wild? We examine what are the common test annotation change patterns in the wild. Studying predominant annotation changes reveals frequent maintainability activities developers perform through test annotation changes as software evolves. Such insights act as stepping stones for our subsequent qualitative study on the motivations and challenges behind test annotation changes.

RQ1: How common are test annotation changes?

We study how frequently developers change test annotations. To provide some comparative statistics, we show the prevalence of test annotation changes compared to common source code transformations (i.e., renames and type changes) at the same program element level (i.e., class, method and field declaration). In particular, we compare test annotation changes at the method level with Rename Method and Change Return Type, at field level with Rename Field and Change Field Type, and at the class level with Rename Class. Such a comparison is attainable because all compared changes are performed on the same kind of program elements. We used the tool implemented by Ketkar et al. [29] to detect renames and type changes. Ketkar et al. report an average precision of 99.7% and a recall of 94.8% for type change detection, and an average precision of 99% and recall of 91% for rename detection, which is very close to the precision/recall values reported in Table II, allowing for a fair comparison annotation change and refactoring practices.

Table V compares the prevalence of test annotation changes with that of the refactoring changes in test code from the 82,810 commits. As shown in Table V, the number of test annotation changes is comparable to the number of test refactorings (i.e., 26.5% difference). Test annotation changes are performed at a method and class level more than renames and type changes, i.e., 91.3% and 137.2% more, respectively. However, at the field level, test annotation changes occurred less than renames and type changes. Despite the popularity of test annotation changes, little tool support exists for test annotations compared to common code transformations such as renames and type changes.

We find that much fewer commits modify test annotations than those that perform renames and type changes. Out of the 82,810 commits, 5,249 commits (6%) modify test annotations, and 17,914 (21%) perform code transformations such as renaming and type changes. Once normalized by the number

TABLE VI
QUANTITATIVE ANALYSIS: TOP THREE HIGHEST FREQUENCY OF ANNOTATION ADDITION, REMOVAL AND MODIFICATION.

Addition	Freq.	**Removal**	Freq.	**Modification**	Freq.
Field Level					
@Mock	147	@Rule	57	@Mock	23
@Rule	42	@Mock	31	@Parameter	12
@Parameter	23	@ClassRule	19	@Parameterized	4
Method Level					
@Ignore	1362	@Ignore	968	@Test	6874
@Before	1238	@Before	482	@Parameterized	94
@After	584	@BeforeClass	293	@Parameters	25
Class Level					
@RunWith	770	@Ignore	318	@Category	1506
@Category	734	@RunWith	306	@RunWith	326
@Ignore	482	@Category	201	@PrepareForTest	91

TABLE VII
QUANTITATIVE ANALYSIS OF ANNOTATION REPLACEMENTS.

Granularity	Annotation Changes	Freq.	Total
Field Level			**27 (2.4%)**
JUnit	@Rule ↔ @ClassRule	17	
JUnit4 → JUnit5	@ClassRule/@Rule → @RegisterExtension	4	
	@ClassRule → @Container	3	
	@Rule → @TempDir	3	
Method Level			**1,007 (91%)**
JUnit	@BeforeClass/@AfterClass → @Before/@After	332	500 (45%)
	@Before/@After → @BeforeClass/@AfterClass	148	
	@Before/@After → @After/@Before	11	
	@Parameters ↔ @Parameterized	6	
	Timeout=X in @Test → @Timeout	2	
	@BeforeClass → @AfterClass	1	
JUnit4 → JUnit5	@Before → @BeforeEach	216	313 (28%)
	@After → @AfterEach	43	
	@BeforeClass → @BeforeAll	23	
	@AfterClass → @AfterAll	19	
	@Ignore → @Disabled	12	
TestNG → JUnit	@BeforeMethod/@AfterMethod → @Before/@After	108	142 (13%)
	@Test(Enabled=False) → @Ignore	30	
	@BeforeTest → @Before	4	
Custom ↔ JUnit	@TestTag → @Category	25	27 (2.5%)
	@Category(PerformanceTest.class) → @PerformanceTest	2	
TestNG	@BeforeTest → @BeforeClass/@BeforeMethod	6	21 (1.9%)
	@AfterTest → @AfterClass/@AfterMethod	9	
	@AfterClass → @AfterMethod	3	
	@AfterMethod → @BeforeMethod	1	
	@BeforeClass → @BeforeMethod	2	
JUnit4 → SpringBoot	@Category → @IntegrationTest	4	4 (0.4%)
Class Level			**70 (6.3%)**
JUnit4 → JUnit5	@Ignore → @Disabled	46	68 (6.3%)
	@RunWith → @ExtendWith	17	
	@Ignore → @Category	5	
JUnit4 → SpringBoot	@Category → @IntegrationTest	2	2 (<1%)
Total		**1104**	

of commits, test annotation changes at class and method level are performed much more frequently than renames and type changes. This shows that test annotation changes at class and method levels are more concentrated in fewer commits, suggesting that annotations may be associated with dedicated maintenance activities. In RQ3, we will further discuss ways annotations are utilized in the maintenance of test code.

Test annotation changes are comparable to renames and type changes at the same program element level and are even more frequently applied at the method and class level. Despite the popularity, there is currently negligible tool support (i.e., antipattern detection, annotation change suggestions, and annotation API usages) for test annotation changes.

RQ2: How are test annotations changed in the wild?

We present the quantitative analysis on test annotation change patterns from two aspects. First, we present the raw change patterns based on three types of changes: addition, removal and modification. The three types of changes are the direct output from our RefactoringMiner extension. Table VI lists the top three annotation changes per change type at three program levels (i.e., field, class and method). We observe that modification has a strong prevalence at a method and class level across the three types of changes. Developers frequently update parameters of the @Test (e.g., timeout=X) and @Category annotations. In general, we notice that developers frequently change the test annotations from mocking frameworks, i.e., add @Mock and @RunWith (20% of PowerMockRunner, 14% of MockitoJUnitRunner). Considering the low prevalence of mocking frameworks in tests (around 1% as shown in Table III), this suggests that the mocking frameworks are frequently updated as code evolves. We also find that developers frequently add and modify @Parameter, @Parameterized and @RunWith (50% of Parameterized) annotations, suggesting that expanding test input and diversifying test execution settings is commonly leveraged to facilitate code evolution. We also observe a high prevalence of adding and removing the @Ignore annotation at both method and class level (i.e., disabling and enabling test cases and test classes). This suggests that technical debt may occur in test code evolution. Lastly, we observe a large number of modifications for the @Category annotation at the class-level to organize test classes into groups, and a diverse set of fixture additions and deletions (i.e., @Before, @After, @BeforeClass).

Furthermore, we perform an in-depth analysis to reveal a composite change pattern, i.e., an annotation replacement. A replacement @X→@Y occurs when annotation @X is removed and @Y is added on the same program element and commit. Annotation replacements may happen, within one testing framework, or between different testing frameworks. Replacements show that as software evolves, the original test annotation (or framework) does not satisfy the testing needs. Therefore developers may look for alternatives. However, such alternatives are not directly provided or are hard-to-achieve in the current framework. Hence, developers need to compromise with workarounds or adopt another framework. Mining annotation replacements is straightforward, based on the output of our RefactoringMiner extension. Specifically, for each commit, we match pairs of removed and added annotations on the same program elements (i.e., fields, methods, and classes) and ensure that these pairs involve different annotation types.

In total, we mined 1,104 replacements from all mined annotation changes. Table VII shows the frequencies of different replacement patterns. Most (91%) of the replacements are at the method level, and 45% of the replacements are switching between JUnit fixtures. For example, developers replace @BeforeClass/@AfterClass with @Before/@After or vice versa to configure the setup and tear down phases at the test class or test case level. Similarly, developers replace @Rule with @ClassRule at field level to expand the impact of a rule to the entire class. At all program element levels, we notice that many replacements occur due to migrations, i.e., between different testing frameworks, or from JUnit4 to

JUnit5. Interestingly, we observe a few replacements between different frameworks, which are not due to migrations. Developers may find a similar test annotation in SpringBoot more suitable for a particular development need than the JUnit @Category annotation. Another common case is to replace custom annotations with the JUnit ones. Developers may define custom annotations for particular needs as JUnit may not yet support the desired features, or developers may not be aware of such support by JUnit. When the developers become aware of the unused JUnit features, or the desired features are shipped in the next JUnit releases, they tend to replace their custom annotations.

Our study shows that developers modify test annotations more frequently compared to additions and removals. Annotations from mocking frameworks are commonly changed despite their low prevalence. Further analysis of annotation replacements shows that they commonly occur for migrating to newer framework versions or other frameworks.

V. A Qualitative Study on Test Annotations

In this section, we conduct a qualitative study to understand the reasons that developers change annotations by answering the following research question:

RQ3: Why do developers change test annotations? Our goal is to provide suggestions to researchers, practitioners, and framework designers on opportunities to improve test annotations. We derive a taxonomy of test annotation changes representing distinctive test maintenance efforts. We believe our taxonomy will provide insights on the maintenance of test annotations and how to improve test quality.

RQ3: Why do developers change test annotations?

We manually study and understand the reasons that developers change test annotations. Our manual study is composed of the following phases:

Phase I: We use stratified random sampling, with a 95% confidence level and 5% confidence interval, to acquire 368 samples from the test annotation changes identified by our RefactoringMiner extension. We adopted stratified random sampling to sample each studied system independently to reduce sampling error when a sub-population within the overall population varies [30].

Phase II: To create the taxonomy for the test annotations, we first classified the changes at a high level based on the annotation type (e.g., @Ignore). Then, the first two authors of the paper (A1 and A2) independently derived an initial list of the reasons behind annotation changes by manually inspecting the relevant commit messages, test source codes, and bug reports.

Phase III: Authors A1 and A2 unified the derived reasons and compared the assigned reason for each annotation change. Any disagreement was discussed until reaching a consensus. The inter-rater agreement of the coding process has a Cohen's kappa of 0.91, indicating almost a perfect agreement level [31].

Table VIII shows the derived taxonomy of the reasons that developers changed the *JUnit* annotations (upper half of the table) and the annotations from other frameworks (lower half of the table) that we found in the sample. To encourage the replication of our results, we have made the dataset available.[1]

JUnit Test Annotation Changes

@Ignore (32%). @Ignore is the most frequently changed test annotation (mostly added). Developers often use this annotation to temporarily disable the execution of tests when there are software bugs or flaky tests. Developers may also bypass test failures caused by recent code changes during a feature addition that breaks a test. For example, in *Hive* (7f4a3e17), the developer ignored the failing test code due to breaking changes during feature addition to pass the test temporarily. Other instances of adding @Ignore are due to dependencies with external libraries. For example, developers disabled a test while waiting for a new software version (e.g., JDK update). However, developers may also add @Ignore to replace automated testing with manual testing when the test code requires manual startup. Although developers frequently ignore tests to facilitate maintenance difficulties, this practice may become ad-hoc and affect code quality. We found instances where developers use @Ignore to pass failing tests without fixing the issue in the code. For example, in *Druid* (da32e1ae), the developer disabled a test due to unknown failure. Later on, the bug persisted, but the test was enabled, and the issue was closed. Similarly, in *Camel* (8ba68e34), the developer disabled a test due to external dependencies during feature addition. However, the ignored test is never enabled. We further conducted an exploratory investigation to see whether the ignored flaky tests in Table VIII are fixed and later enabled. We find that developers often do not find the root cause of test flakiness and ignore the test in the entirety of software evolution, indicating that ignored tests persist and are often forgotten.

As @Ignore becomes a common way to bypass challenges in test maintenance, it may become ad-hoc and a source of technical debt.

@Test(Timeout=X) (14%). Our analysis reveals innovative uses of timeout and maintenance problems of such uses due to software's ever-evolving nature. We find that timeout is employed to achieve various goals, i.e., detecting deadlocks (e.g., *Hbase-2428c5f*) and performance regressions (e.g., *Hadoop-f131dba8*), and providing meaningful debugging information when accessing external resources. While timeout is an effective tactic to serve the aforementioned purposes, we observe that developers constantly need to increase the timeout threshold (even by removing the use of timeout entirely) to avoid test failures and to accommodate the evolving software development. For example, once the running environment changes (e.g., to a slower cluster or platform), test execution may become slower and lead to timeout errors. Developers need to increase the timeout threshold to avoid test failures.

[1] https://github.com/SPEAR-SE/TestAnnotationMaintenance_Data

TABLE VIII
QUALITATIVE ANALYSIS: TAXONOMY OF ANNOTATION CHANGES.

Annotation Type	Motivation	Frequency
	JUnit Test Annotations Changes	
Ignore		**115**
Bugs	Adding @Ignore to bypass test failure caused by bugs in test/source code.	42
Flaky test	Adding @Ignore to disable flaky tests.	33
External dependency	Adding @Ignore when an external dependency needs to be manually configured (e.g., database), or the developers wait for a new release of an external dependency to resolve an issue (e.g., JVM).	20
Feature addition/improvement	Adding @Ignore to disable tests that are related to incomplete features/code changes.	19
Timeout		**51**
Relax timeout	Increasing @Test(timeout) thresholds to accommodate slow cluster, slow machine or slow tests.	23
Deadlock detection	Adding @Test(timeout) to help detect deadlocks (i.e., if tests do not finish within the specified time, there may be a deadlock).	15
External resource retrieval	Adding @Test(timeout) to complement tests that retrieve an external resource. For example, without a timeout, the test may fail due to `NullPointerException` and suppress the actual fault (e.g., resource unavailability).	6
Perf. regression detection	Adding @Test(timeout) to ensure that the test finishes on time for detecting performance regression.	6
Ad-hoc timeout removal	Removing @Test(timeout) completely as tests become too slow instead of relaxing the timeout.	1
Fixture		**35**
Reset fixtures	Replacing @BeforeClass with @Before to reset fixtures for each test case (e.g., for bug fixing or test case isolation).	14
Improve test speed	Replacing @Before with @BeforeClass to improve test time by removing unnecessarily repeated fixture initialization.	10
Inflexible configuration	Removing or changing fixtures since they are not configurable per test case (e.g., @Before method runs for every test case, while @BeforeClass runs only once before a test case).	6
Maintainability	Adding fixtures to remove duplicate initialization in the test code for better maintainability.	1
Category		**37**
Test prioritization	Adding @Category to group tests based on their speed/size to detect failures more quickly (e.g., run faster tests first).	33
Ignore tests	Adding @Category in addition to @Ignore to organize ignored tests for future maintainability.	4
Parameterized		**31**
Increase test coverage	Adding or changing @Parameterized to increase coverage for failing corner cases, or newly added features.	16
Refactor test code	Adding @Parameterize to refactor tests to improve maintainability (e.g., share common test inputs or test code).	7
Parallelize tests	Adding @Parameterized and use a thread pool to run tests in parallel and speed up test execution.	4
Add debugging messages	Changing @Parameterized parameter to include optional messages for improved debugging.	3
Slow test	Removing @Parameterized to improve test execution time. In JUnit4, @Parameterized is limited to class level. If only subsets of tests use parameterized annotations, then it may increase test execution time.	1
Expected Exception		**18**
Adjusting exception handling	Changing between different exception handling mechanisms (i.e., *JUnit3* Try with fail, *JUnit4* @Rule, *JUnit5* Assertions.assertThrows, *JUnit4* @ExpectedException, or even custom expected exception)	14
Test Driven Development	Adding expected exception to complement test-driven development by making the test pass with known exceptions until the feature implementation is done.	2
Exception too general	Changing expected exception from a general exception type to a more specialized one. For example, @ExpectedException(GenericException) can pass the expected exception test, but does not provide details about the actual exception type.	2
Rule		**14**
Refactor via @Rule	Adding built-in or custom (e.g., extract duplicate fixture) @Rule to improve test code maintainability.	14
Fixed Test Order	Adding @FixMethodOrder from *JUnit4* to enforce deterministic test orders and fix flaky tests.	**5**
	Other Types of Test Annotations Changes	
Migration		**21**
JUnit4 to *JUnit5* migration	Manual migration from *JUnit4* to *JUnit5* (e.g., one package at a time), resulting in sparse *JUnit5* adoption. Typically migrated annotations are related to fixtures (i.e., @Before to @BeforeEach), and sometimes from @RunWith to @ExtendWith, or from @Category to @Tags.	19
JUnit3 to *JUnit4* migration	Automated migration from *JUnit3* to *JUnit4* using tool support.	1
TestNG to *JUnit5*	Remove TestNG in favour of *JUnit5* due to its popularity.	1
Mocking		**17**
Namespace error in mocking	Mocking frameworks, such as PowerMock, utilize an independent classloader, which may cause namespace error (i.e., class not found).	11
Mock usage	Adding Powermock to mock final, utility and abstract class.	5
Mock *vs* Injection	Removing mocking and use dependency injection instead.	1
Custom		**11**
Retry on failure/exception	Implementing custom annotations to retry test on failure (*JUnit4*). Developers should leverage *JUnit5* @RepeatedTest	8
Experimental	Implement custom annotation to categorize tests during feature addition to detect untested code, instead of using *JUnit4* @Category.	3
Mixed Framework Usage		**3**
Using @DependsOn in *TestNG*	Using *TestNG* in addition to *JUnit*, because *TestNG* provides the @DependsOn annotation, which enforces test ordering. However, developers can leverage *JUnit4* @FixMethodOrder to avoid using multiple frameworks.	2
Using @Test(group=X) in *TestNG*	Using *testNG* in addition to *JUnit*, because *TestNG* provides the group option inside @Test annotation, which categorizes tests. However, developers can leverage *JUnit4* @Category or *JUnit5* @Tag to avoid using multiple frameworks.	1
By-product	Changing/adding/removing test annotation due to feature deletion or test code relocation.	**10**

Another example is that detecting performance regressions may become flaky as code evolves, i.e., the execution time comes closer to the timeout threshold and leads to unstable test results in different runs. To avoid flakiness, developers may increase the timeout threshold. This shows that timeout may not be suitable for performance testing, and a framework (e.g., OpenJDK JMH) that allows more sophisticated settings (e.g., repeated runs, warm-up iterations) should be leveraged.

> Developers use @Test(timeout=X) to detect concurrency issues or performance regressions; however, they may also relax/remove the timeout when performance regressions occur.

Fixtures (9.6%). Our fixture change analysis reveals the inherent difficulties and error-proneness in maintaining the balance between a clean test environment and minimized test execution time. To minimize test execution time, developers may continuously refactor test initialization code, i.e., extracting duplicate initialization code in a separate method with fixture annotations, or changing a fixture method from @Before to @BeforeClass (*Druid-da32e1ae*). However, some fixture code (e.g., resetting shared variables), if incorrectly placed in a class fixture (i.e., @BeforeClass), may violate the test independence assumption [32], introduce bugs in test code, and produce unstable test results (i.e., flaky test). Therefore, developers may perform changes to execute such a fixture code for each test method repeatedly (e.g., *Ambari-7153112e*). Interestingly, we noticed that developers expressed performance concerns on such changes and sometimes, they were even uncertain about whether such changes would completely fix the buggy

test. Hence, developers may benefit from having a detection tool that helps them determine the trade-off (e.g., a search-based approach). Moreover, we observe that developers need to perform workarounds due to the limitations of expressing fixtures in JUnit. In particular, developers may remove JUnit fixture to resort to a direct call to the parameterized helper methods, increasing redundancies. This happens when there is a need to adopt different fixtures for each test method. However, JUnit does not support tailoring fixture, i.e., all the test cases in one test class share the same fixture. Another limitation is the lack of fine-tuning JUnit fixtures based on the test cases. For example, in *UpdateActiveRepoVersionOnStartupTest - Ambari* (2700bd125f), developers replaced JUnit @Before with a parameterized helper method to conditionally configure the cluster in the fixture based on test cases. We find that developers performed such workarounds due to JUnit limitations complicate test fixture and may increase test maintenance overhead.

Developers are concerned about the tradeoff between minimizing test code duplication and minimizing test execution time. Furthermore, the lack of configuration capabilities in JUnit fixtures causes developers to perform workarounds, increasing technical debt and maintenance overhead.

@Category (10%). We find that developers mostly add @Category to categorize tests for test prioritization. In *Hbase*, the category groups test based on timeout thresholds. Developers acknowledge that categorizing based on timeout can improve regression testing practice by running faster tests first (i.e., the ones with smaller timeout threshold) to detect bugs quickly. We also find that developers add @Category to complement ignored tests for better maintainability. For instance, a "FailingTest" category can indicate ignored tests due to test failure. Developers may also use @Category to specify whether certain tests are integration tests or unit tests.

Developers customize @Category to assist diverse maintenance needs. We find that categorization based on test execution time can be useful for efficient regression test analysis, and can also help detect failed or ignored tests for more reliable maintainability.

@Parameterized (8.5%). We find that most changes to @Parameterized align to its regular use, i.e., increase the flexibility of managing test inputs. Developers may add more corner cases to the input using @Parameterized after adding new features or fixing bugs to avoid regression. We also find that developers may refactor the test code using @Parameterized. We find cases where developers use @Parameterized to reuse common test input arguments in different test cases. For example, in a test from *Hadoop* (ad1b988a8286), developers use @Parameterized to remove multiple subclasses that share the same code with only differences in test input. We also find that developers may use @Parameterized to run tests with different inputs in parallel to speed up test execution. However, one limitation of @Parameterized is that it can only be applied at the class level. Therefore, if only a subset of the test methods needs the parameterized annotation, there may be additional setup overhead that slows down the test. Finally, the JUnit4 Parameterized class contains an optional string pattern that helps decorate test results with additional messages for improved debugging. Including such messages is considered the best practice by some developers, as we found in our manual study. Nevertheless, most of our manual study samples do not leverage this best practice, and thus we believe users of JUnit4 can benefit from knowing about it.

@Parameterized is used to enable test code to take arguments and refactor test code and test input duplications. Moreover, developers can improve the test execution speed of a @Parameterized test by using parallel programming.

Expected Exception (4.9%). Developers sometimes need to test if the code would throw a specified exception for erroneous behaviours. Over the JUnit history, encoding expected exceptions in test cases can be achieved differently, i.e., *try* with *fail* in JUnit3, @Rule and @ExpectedException in JUnit4, and *assertThrows* in JUnit5. On the one hand, we find that new JUnit releases gradually adapt to the increasing need for handling expected exceptions elegantly, i.e., a specialized annotation @ExpectedException instead of the workaround *try* with *fail*, and an improved annotation *assertThrows* to overcome the limitation of @ExpectedException. On the other hand, we find that developers may not be fully aware of new features in their adopted JUnit version.

For example, we find cases that developers may migrate back to JUnit3 to use the *try* with *fail* mechanism, because @ExpectedException is limited in providing comprehensive error messages. Such a backward migration is not needed, since developers could leverage @Rule from JUnit4, or migrate to JUnit5 `assertThrows`. In other cases, we find that developers tried to customize their test code to handle expected exceptions, but later migrated to a more framework-dependent pattern as described above. *Our findings suggest a potentially ill-defined knowledge of how to handle expected exceptions in practice. Developers could benefit from having an expected exception recommendation tool to reduce test maintenance overhead*. We also find that developers may use expected exceptions to facilitate test-driven development. Namely, the expected exception avoids test failures while developers actively work on implementing the features. Finally, we find a misuse of the expected exception associated with the *Exception* type. Developer discussions reveal that the use of general excepted exception types (e.g., `java.lang.Exception`) could hide the actual faults because the test will still pass if an unexpected sub-type exception is thrown. Therefore, one way to solve this issue is to use specific exception types instead of a general exception type.

There is a diverse way to handle expected exception in test code. Developers sometimes are unaware of which mechanism to utilize and what mechanisms may be available in their adopted JUnit version.

@Rule (3.8%). JUnit4 introduced @Rule to provide a flexible

mechanism to enhance tests by running code around a test execution, similar to @Before and @After. In alignment with the regular use of the @Rule, we find that developers often refactor duplicate test code using @Rule to improve maintainability and readability. We also found three common uses of built-in @Rules, namely the Timeout, TemporaryFolder, and TestName rules [33]. In these cases, developers preferred using these built-in @Rules to simplify test code. Future studies may consider using various annotations to help refactor test code.

> Developers utilize @Rule to remove duplicate code in test fixtures. Moreover, we find that developers can benefit from using built-in @Rules to simplify their test code.

@FixMethodOrder (1.4%). Prior studies [3], [34]–[36] found that one of the root causes of flaky tests is test order dependencies. JUnit4 provides @FixMethodOrder to allow a deterministic test execution order. For example, we find in *Camel* (4e7ec8f79b6) that since JDK7 does not preserve test execution order, developers fixed test flakiness using @FixMethodOrder. Hence, there is a future research opportunity on automated test fixing by applying such annotations.

> Developers apply @FixMethodOrder to ensure a deterministic test execution order and avoid dependency-related flaky tests.

Other Types of Test Annotations Changes

Migration (5.8%). In the migration from JUnit3 to JUnit4, we find that developers use an automated tool that applies migration in the entire codebase. However, the migration from JUnit4 to JUnit5 is applied manually and slowly (e.g., one package at a time). There are many migrations that started over one year ago (i.e., before 2019), but the issue reports still remain open today. We believe migrations to JUnit5 are intentionally manual, because some annotations, such as @Rule, are removed in JUnit5. Moreover, some annotations are renamed and may further cause confusion to developers (e.g., @Before is renamed to @BeforeEach). Therefore, developers could benefit from having an automatic JUnit5 migration tool. Finally, we find some changes where developers migrate from TestNG to JUnit5 due to the popularity of JUnit.

> We find that the migration from JUnit4 to JUnit5 is done manually and slowly. To help developers utilize the new features in JUnit5, future studies should further investigate migration patterns for JUnit5 in order to assist developers with automated migration.

Mocking (4.7%). A JUnit class annotated with @RunWith indicates that the JUnit framework invokes a specified class using a developer-specified test runner instead of running the default runner. An issue with using mocking runners is that these mocking frameworks utilize an independent class loader, which sometimes causes namespace error due to conflicting classes. To resolve the issue, developers add @PowerMockIgnore to defer loading the conflicting classes. Finally, for one instance, we find that developers decided to remove mocking and use dependency injection instead.

> Developers often use mocking for external dependencies. However, they may encounter issues related to namespace conflicts.

Custom Annotation (3%) and Mixed Framework Usage (0.8%). We found cases where developers created customized annotations to repeat the test execution upon failure (e.g., for detecting flaky test), or to indicate that a test is experimental. However, JUnit5 provides a new annotation @RepeatedTest, and both JUnit4 and JUnit5 provide annotations (e.g., @Category) to categorize tests. We also find cases where developers added annotations from TestNG, even though developers were already using JUnit, which provides similar annotations. The findings may indicate that sometimes developers might not know the functionality that is provided by testing frameworks, and may use customized annotations and increase maintenance costs.

> Developers resort to customized annotations due to the lack of awareness about the features offered by testing frameworks, suggesting the need for annotation recommendation tool support.

By-product (2.7%). We find that developers may modify test annotations due to a feature removal or test refactoring (e.g., relocate the annotation to another test file).

VI. Implications and Future Work

Based on our empirical findings, we present actionable implications and future work for three groups of audiences: 1) researchers, 2) application developers and testers, and 3) framework designers.

A. Researchers

R1: Test annotation is an integral part of test design and implementation. Future studies on test maintenance and refactoring should consider the peculiarity of test annotations. As we find in RQ1, test annotation changes are frequent in test maintenance, and developers often use test annotations to improve test maintenance. As we found in RQ3, developers use various test annotations, such as fixtures or @Rule, to remove duplication in test code. Based on our manual study, such refactorings are commonly performed, but there is a lack of tool support. Therefore, future refactoring studies may want to design test code refactoring techniques that leverage such test annotations.

R2: Developers may use test annotation for ad-hoc fixes, which may affect test maintenance or even code quality. Future studies are needed to study such impact and provide research solutions. In RQ3, we find that developers may use test annotations for ad-hoc fixes (e.g., adding @Ignore to failing tests or increasing the timeout threshold in @Test(timeout=X) without finding the root cause). Although the fixes make the tests pass, the underlying issues remain unsolved, which may cause more severe issues in the future. Moreover, as we found in RQ2, there are much more additions of @Ignore than removals, which indicates that many tests get disabled without being re-enabled. Future studies are needed

to study the prevalence of such technical debt in ad-hoc test fixes and their potential consequences.

R3: There are future research opportunities on detecting misuses of test annotations. Recently, test smell detection starts to receive interest from both the academia and industry due to its practicality [5], [8], [12], [13], [37], [38]. However, most prior studies only consider test smells related to test code, yet as we found in RQ3, there exist many test annotation misuses. For example, we found that developers use suboptimal ways to handle expected exceptions or @Parameterized in tests (e.g., tests expecting generic exceptions or not recording error messages when using @Parameterized). Furthermore, there are also possible test annotation misuses related to fixtures (e.g., using @Before without considering its performance impact). Our study highlights and opens an avenue for future research to better detect test smells and improve test quality.

R4: Future research is needed to provide automated test grouping and better utilization of test annotations to reduce test execution overhead. To reduce test execution overhead, prior studies have proposed various test selection [39]–[42] and prioritization techniques [43]–[50]. In RQ3, we find that developers use @Category to group and execute small tests first (e.g., run faster tests first to detect failures early). Future studies may consider integrating their techniques with test annotations for better research adoption. Developers may also use parallelization to speed up test execution (e.g., using @Parameterized with thread pools). However, we find that JUnit5 provides a new annotation, @Execution(ExecutionMode.CONCURRENT), which allows parallel test execution. Future study is needed to assist developers to automatically adopt such annotations and improve test execution time without causing concurrency issues or flaky tests.

B. Application Developers and Testers

A1: Developers need better education about the capabilities of testing frameworks. As found in a prior study [51], due to differences in background, some developers may not be familiar with specific frameworks. In RQ3, we also have similar observations with testing frameworks. Even though JUnit is the most commonly used framework in Java [9], we find that some developers do not fully utilize test annotations. For example, some developers were unsure which way they should use to handle expected exceptions, and some developers created custom test annotations, even though JUnit already provides the same functionality. A prior study found that there is often a champion who first adopts new features in a framework and helps the team with adoption [23]. We recommend that developers follow similar procedures and dedicate at least one team member to gain expertise in testing frameworks and help the development team utilize testing frameworks.

C. Framework Designers

F1: Framework designers need to provide better flexibility in their APIs. In RQ3, we find some cases where developers need to find workarounds or adopt other testing frameworks to bypass some inflexibility in JUnit. For example, @Before is executed for every test case in the class, but developers may only want @Before to be executed for a subset of the test cases. In this case, developers removed JUnit fixture annotations and replaced them with a direct call to the helper method, increasing redundancies. In other cases, fine-tuning fixtures based on test cases also enforced developers to use a Parameterized helper method over JUnit @Before. Finally, we also uncovered some limitations with @Parameterized in JUnit4, although it can remove test code and test input duplication and improve test coverage. Firstly, the JUnit4 parameterized class only works at class level, and cannot be configured at method level. Thus, for the specified test inputs, the test runner will execute every single test case even if not all test cases utilize the input. Hence, we believe that software engineering researchers may work with framework developers and identify possible issues that framework users encounter and improve the framework accordingly.

F2: Better annotation support targeting specific testing issues (e.g., flaky tests). We find that JUnit provides annotations, such as @FixMethodOrder, to resolve issues related to order dependent flaky tests. Similarly, one of the customizations we find is retrying on test failure. However, JUnit5 now provides a new annotation @RepeatedTest to help developers detect test flakiness more easily. To this end, with the recent research advances in the detection of flaky tests, we believe that incorporating more support in a practical framework, such as JUnit, will help developers quickly address test flakiness without resorting to other specialized tools that are difficult to adopt in practice.

VII. Threats to Validity

Internal Validity. Our findings depend on the accuracy of our tool to mine annotation changes from the commit history. We mitigate this threat by validating our tool thoroughly. The extension of RefactoringMiner 2.0 detects annotation changes with a 99.7% precision and 98.7% recall.

External Validity. We study systems that are all open source implemented in Java, so the result may not be generalizable to all systems. To minimize the threat, we follow a set of criteria to select systems that are popular on GitHub, large in scale, and actively maintained. The studied systems cover various domains and are frequently used in commercial settings.

Construct Validity. We conduct a manual study to understand the reasons behind test annotation changes. Due to the large number of changes, we take a statistically significant sample. There may be bias or misidentification in our manual study on characterizing test annotation changes. Thus, two authors independently examined all available software artifacts and discussed them until the agreement is made. We do not claim to find all usage and misusage patterns, and the limitations of test annotations. However, we show the existence of such patterns and identify further research opportunities. Thus, fu-

ture work should survey developers based on recent annotation changes to gain additional insights.

VIII. Conclusion

This paper presents the very first empirical study on annotation changes in Java tests to fill the knowledge gap regarding the evolution and maintenance of tests, since prior studies focused mainly on the test code and ignored the test annotations. Our study reveals many interesting findings with actionable implications:

1) Test annotation changes are more common than test refactorings. Despite that, there is very limited tool support for migrating test annotations to newer framework versions or different frameworks, and automating common annotation change patterns within the same framework version.
2) Test developers are sometimes *unaware of the features provided by testing frameworks*, and thus apply alternative suboptimal solutions. There is great need for tool support to detect the misuse (or lack of use) of annotations and recommend appropriate test annotations.
3) Test developers are forced to apply workarounds to overcome the current *limitations of testing frameworks*. Framework designers need to be aware of these workarounds to improve the design and flexibility of test annotations.

References

[1] N. B. Ali, E. Engström, M. Taromirad, M. R. Mousavi, N. M. Minhas, D. Helgesson, S. Kunze, and M. Varshosaz, "On the search for industry-relevant regression testing research," *Empirical Software Engineering*, vol. 24, no. 4, pp. 2020–2055, 2019. [Online]. Available: https://doi.org/10.1007/s10664-018-9670-1

[2] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 643–653.

[3] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta, "Root causing flaky tests in a large-scale industrial setting," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, D. Zhang and A. Møller, Eds. ACM, 2019, pp. 101–111. [Online]. Available: https://doi.org/10.1145/3293882.3330570

[4] A. van Deursen, L. Moonen, A. van den Bergh, and G. Kok, "Refactoring test code," in *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*, 2001, pp. 92–95.

[5] A. Peruma, K. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba, "On the distribution of test smells in open source android applications: An exploratory study," in *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, ser. CASCON '19, 2019, pp. 193–202.

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

[7] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, "An empirical analysis of the distribution of unit test smells and their impact on software maintenance," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, 2012, pp. 56–65.

[8] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, and A. Bacchelli, "On the relation of test smells to software code quality," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 1–12.

[9] A. Zerouali and T. Mens, "Analyzing the evolution of testing library usage in open source java projects," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017, pp. 417–421.

[10] S. Levin and A. Yehudai, "The co-evolution of test maintenance and code maintenance through the lens of fine-grained semantic changes," in *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*. IEEE Computer Society, 2017, pp. 35–46. [Online]. Available: https://doi.org/10.1109/ICSME.2017.9

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

[12] N. S. Junior, L. R. Soares, L. A. Martins, and I. Machado, "A survey on test practitioners' awareness of test smells," *CoRR*, vol. abs/2003.05613, 2020. [Online]. Available: https://arxiv.org/abs/2003.05613

[13] A. Qusef, M. O. Elish, and D. W. Binkley, "An exploratory study of the relationship between software test smells and fault-proneness," *IEEE Access*, vol. 7, pp. 139 526–139 536, 2019. [Online]. Available: https://doi.org/10.1109/ACCESS.2019.2943488

[14] V. Garousi and B. Küçük, "Smells in software test code: A survey of knowledge in industry and academia," *Journal of Systems and Software*, vol. 138, pp. 52–81, 2018. [Online]. Available: https://doi.org/10.1016/j.jss.2017.12.013

[15] N. Tsantalis, A. Ketkar, and D. Dig, "Refactoringminer 2.0," *IEEE Transactions on Software Engineering*, 2020.

[16] L. S. Pinto, S. Sinha, and A. Orso, "Understanding myths and realities of test-suite evolution," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: Association for Computing Machinery, 2012. [Online]. Available: https://doi.org/10.1145/2393596.2393634

[17] A. Zaidman, B. V. Rompaey, A. van Deursen, and S. Demeyer, "Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining," *Empir. Softw. Eng.*, vol. 16, no. 3, pp. 325–364, 2011. [Online]. Available: https://doi.org/10.1007/s10664-010-9143-7

[18] D. D. Ma'ayan, "The quality of junit tests: an empirical study report," in *Proceedings of the 1st International Workshop on Software Qualities and Their Dependencies, SQUADE@ICSE 2018, Gothenburg, Sweden, May 28, 2018*, S. Sentilles, B. W. Boehm, C. Trubiani, X. Franch, and A. Koziolek, Eds. ACM, 2018, pp. 33–36. [Online]. Available: https://doi.org/10.1145/3194095.3194102

[19] G. Kudrjavets, N. Nagappan, and T. Ball, "Assessing the relationship between software assertions and faults: An empirical investigation," in *17th International Symposium on Software Reliability Engineering (ISSRE 2006), 7-10 November 2006, Raleigh, North Carolina, USA*. IEEE Computer Society, 2006, pp. 204–212. [Online]. Available: https://doi.org/10.1109/ISSRE.2006.14

[20] Y. Zhang and A. Mesbah, "Assertions are strongly correlated with test suite effectiveness," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 214–224. [Online]. Available: https://doi.org/10.1145/2786805.2786858

[21] H. Rocha and M. T. Valente, "How annotations are used in java: An empirical study," in *Proceedings of the 23rd International Conference on Software Engineering & Knowledge Engineering (SEKE'2011), Eden Roc Renaissance, Miami Beach, USA, July 7-9, 2011*. Knowledge Systems Institute Graduate School, 2011, pp. 426–431.

[22] R. Dyer, H. Rajan, H. A. Nguyen, and T. N. Nguyen, "Mining billions of AST nodes to study actual and potential usage of java language features," in *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, P. Jalote, L. C. Briand, and A. van der Hoek, Eds. ACM, 2014, pp. 779–790. [Online]. Available: https://doi.org/10.1145/2568225.2568295

[23] C. Parnin, C. Bird, and E. R. Murphy-Hill, "Adoption and use of java generics," *Empirical Software Engineering*, vol. 18, no. 6, pp. 1047–1089, 2013. [Online]. Available: https://doi.org/10.1007/s10664-012-9236-6

[24] Z. Yu, C. Bai, L. Seinturier, and M. Monperrus, "Characterizing the usage, evolution and impact of java annotations in practice," *IEEE Transactions on Software Engineering*, 2019.

[25] M. R. Marri, Tao Xie, N. Tillmann, J. de Halleux, and W. Schulte, "An empirical study of testing file-system-dependent software with mock objects," in *2009 ICSE Workshop on Automation of Software Test*, 2009, pp. 149–153.

[26] S. Mostafa and X. Wang, "An empirical study on the usage of mocking frameworks in software testing," in *2014 14th International Conference on Quality Software*, 2014, pp. 127–132.

[27] D. Spadini, M. Aniche, M. Bruntink, and A. Bacchelli, "Mock objects for testing java systems," vol. 24, no. 3, p. 1461–1498, Jun. 2019.

[28] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: ACM, 2014, pp. 313–324.

[29] A. Ketkar, N. Tsantalis, and D. Dig, "Understanding type changes in java," in *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE '20, Nov 2020.

[30] X. Zhao, J. Liang, and C. Dang, "A stratified sampling based clustering algorithm for large-scale data," *Knowl. Based Syst.*, vol. 163, pp. 416–428, 2019. [Online]. Available: https://doi.org/10.1016/j.knosys.2018.09.007

[31] A. J. Viera, J. M. Garrett *et al.*, "Understanding interobserver agreement: the kappa statistic," *Family Medicine*, vol. 37, no. 5, pp. 360–363, 2005.

[32] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin, "Empirically revisiting the test independence assumption," ser. ISSTA 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 385–396. [Online]. Available: https://doi.org/10.1145/2610384.2610404

[33] JUnit team. (2018, Jun.) Rules. [Online]. Available: https://github.com/junit-team/junit4/wiki/Rules

[34] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "idflakies: A framework for detecting and partially classifying flaky tests," in *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019*. IEEE, 2019, pp. 312–322. [Online]. Available: https://doi.org/10.1109/ICST.2019.00038

[35] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, S. Cheung, A. Orso, and M. D. Storey, Eds. ACM, 2014, pp. 643–653. [Online]. Available: https://doi.org/10.1145/2635868.2635920

[36] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, "Understanding flaky tests: the developer's perspective," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, M. Dumas, D. Pfahl, S. Apel, and A. Russo, Eds. ACM, 2019, pp. 830–840. [Online]. Available: https://doi.org/10.1145/3338906.3338945

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

[38] N. S. Junior, L. R. Soares, L. A. Martins, and I. Machado, "A survey on test practitioners' awareness of test smells," *CoRR*, vol. abs/2003.05613, 2020. [Online]. Available: https://arxiv.org/abs/2003.05613

[39] M. Gligoric, L. Eloussi, and D. Marinov, "Practical regression test selection with dynamic file dependencies," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 2015, pp. 211–222.

[40] A. Shi, T. Yung, A. Gyori, and D. Marinov, "Comparing and combining test-suite reduction and regression test selection," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015, 2015, pp. 237–247.

[41] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," *ACM Transactions on Software Engineering Methodology*, vol. 6, no. 2, pp. 173–210, 1997.

[42] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid, "An empirical study of junit test-suite reduction," in *Proceedings of the IEEE 22nd International Symposium on Software Reliability Engineering*, ser. ISSRE '11, 2011, pp. 170–179.

[43] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, and B. Xie, "Learning to prioritize test programs for compiler testing," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17, 2017, pp. 700–711.

[44] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *IEEE Transactions on Software Engineering*, vol. 33, no. 4, pp. 225–237, 2007.

[45] H. Mei, D. Hao, L. Zhang, L. Zhang, J. Zhou, and G. Rothermel, "A static approach to prioritizing junit test cases," *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1258–1275, Nov. 2012.

[46] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 10, pp. 929–948, 2001.

[47] R. K. Saha, L. Zhang, S. Khurshid, and D. E. Perry, "An information retrieval approach for regression test prioritization based on program changes," in *Proceedings of the 37th International Conference on Software Engineering*, ser. ICSE '15, 2015, pp. 268–279.

[48] S. W. Thomas, H. Hemmati, A. E. Hassan, and D. Blostein, "Static test case prioritization using topic models," *Empirical Software Engineering*, vol. 19, no. 1, pp. 182–212, Feb. 2014.

[49] S. Yoo, M. Harman, and D. Clark, "Fault localization prioritization: Comparing information-theoretic and coverage-based approaches," *ACM Transactions on Software Engineering Methodology*, vol. 22, no. 3, pp. 19:1–19:29, Jul. 2013.

[50] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei, "Bridging the gap between the total and additional test-case prioritization strategies," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13, 2013, pp. 192–201.

[51] T.-H. Chen, W. Shang, A. E. Hassan, M. Nasser, and P. Flora, "Detecting problems in the database access code of large scale systems: An industrial experience report," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16, 2016, p. 71–80.