How Disabled Tests Manifest in Test Maintainability Challenges?

Dong Jae Kim*
Software PEformance, Analysis and Reliability (SPEAR) Lab
Concordia University
Montreal, Quebec, Canada
k_dongja@encs.concordia.ca

Bo Yang*
Department of Computer Science and Software Engineering
Concordia University
Montreal, Quebec, Canada
b_yang20@encs.concordia.ca

Jinqiu Yang
Department of Computer Science and Software Engineering
Concordia University
Montreal, Quebec, Canada
jinqiu.yang@concordia.ca

Tse-Hsun (Peter) Chen
Software PEformance, Analysis and Reliability (SPEAR) Lab
Concordia University
Montreal, Quebec, Canada
peterc@encs.concordia.ca

ABSTRACT

Software testing is an essential software quality assurance practice. Testing helps expose faults earlier, allowing developers to repair the code and reduce future maintenance costs. However, repairing (i.e., making failing tests pass) may not always be done immediately. Bugs may require multiple rounds of repairs and even remain unfixed due to the difficulty of bug-fixing tasks. To help test maintenance, along with code comments, the majority of testing frameworks (e.g., JUnit and TestNG) have also introduced annotations such as @Ignore to disable failing tests temporarily. Although disabling tests may help alleviate maintenance difficulties, they may also introduce technical debt. With the faster release of applications in modern software development, disabling tests may become the salvation for many developers to meet project deliverables. In the end, disabled tests may become outdated and a source of technical debt, harming long-term maintenance. Despite its harmful implications, there is little empirical research evidence on the prevalence, evolution, and maintenance of disabling tests in practice. To fill this gap, we perform the first empirical study on test disabling practice. We develop a tool to mine 122K commits and detect 3,111 changes that disable tests from 15 open-source Java systems. Our main findings are: (1) Test disabling changes are 19% more common than regular test refactorings, such as renames and type changes. (2) Our life-cycle analysis shows that 41% of disabled tests are never brought back to evaluate software quality, and most disabled tests stay disabled for several years. (3) We unveil the motivations behind test disabling practice and the associated technical debt by manually studying evolutions of 349 unique disabled tests, achieving a 95% confidence level and a 5% confidence interval. Finally, we present some actionable implications for researchers and developers.

CCS CONCEPTS

• **Software and its engineering → Software evolution**; **Maintaining software**.

KEYWORDS

Test Disabling, Test Smell, Test Maintenance, Technical Debt

ACM Reference Format:
Dong Jae Kim*, Bo Yang*, Jinqiu Yang, and Tse-Hsun (Peter) Chen. 2021. How Disabled Tests Manifest in Test Maintainability Challenges?. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3468264.3468609

1 INTRODUCTION

Modern software development handles an increasing complexity of feature enhancements. To ensure that the software quality remains on par with consumer expectations, software testing has been playing a pivotal role in software development. With the availability of JUnit and other testing frameworks, writing test code is becoming widely popular [14, 32]. Most large-scale systems utilize testing practices routinely and expose faults early.

However, test code can be subject to age and quality issues like the production code under test. For example, a test can also contain test-specific design issues that may hinder its ability to guard against regressions. Prior studies [15, 17] have found that flaky tests may hinder the reliability of testing results that may fail for reasons other than recent changes. Similarly, researchers introduced the concept of test smells, which are design issues specific to the test code that may negatively impact test code comprehension and maintenance [3, 29].

To mitigate the efforts to fix broken tests, developers need to maintain and improve test code continuously. Challenges in maintaining and improving test code are more than fixing flaky tests and refactoring test smells. A prior study by Pinto et al. [21] highlights both the importance and potential of studying the evolution of how tests are modified, added, or deleted. Understanding such evolution is essential to understand how the test becomes obsolete, why it is difficult to fix, and how it should be repaired. The findings can inspire future research and provide better testing support and tools.

In the context of software testing, developers can disable tests by commenting out the test method or class. In addition, with the introduction of annotations in Java 5, frameworks such as JUnit and TestNG introduce the annotations @Ignore and @Test(enabled=false), allowing developers to disable failing tests temporarily. Although disabling tests can be seen as an added flexibility for developers to alleviate maintenance difficulties, one can suspect that it may introduce technical debt. With such flexibility, developers may disable hard-to-fix tests as a compromising solution to meet

* Dong Jae Kim and Bo Yang contributed equally to this paper.

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08…$15.00
https://doi.org/10.1145/3468264.3468609

project deliverables. Despite the potential challenges that arise later from disabling tests, there exist limited studies on disabled tests as a source of technical debt. We believe that studying why developers disable test code is of paramount importance for both practitioners and researchers: (1) it indicates a source of potential technical debt that can direct future research efforts, (2) it may provide additional evidence on how bugs are fixed to improve automatic tool support, and (3) can help prevent future encounters of bugs.

To the best of our knowledge, there are no studies in the literature investigating disabled tests as a source of technical debt and providing tools for tracking all types of disabling changes in the test code. To address this issue, we develop an automated tool to identify all kinds of disabling and re-enabling practice at commit level: (1) a commented-out test code instance, (2) commenting out or deleting @Test, (3) using @Ignore from JUnit, and (4) setting @Test(enable=false) in TestNG. Our approach could detect the disabling/re-enabling practices with an overall precision of 96%.

In total, we study test disabling practices in 15 open-source systems of different sizes and from diverse domains. Our study focuses on understanding the disabled tests from the following aspects: how often do developers disable a test (i.e., the prevalence and evolution) and why developers disable a test (i.e., motivations behind disabling and re-enabling a test). In particular, we answer the following three research questions (RQs):

RQ1: How common are test disabling changes? Test disabling practices are 19% more common than regular test refactorings, such as renames and type changes. Even though we find that disabling tests are prevalent, there is no prior study on how they may affect test maintenance.

RQ2: What is the change pattern of disabled tests? Through analyzing the change pattern and final destination of the disabled tests, we find that most disabled tests stay disabled. Many of the disabled tests have been disabled for several years. We also find that for the disabled tests that are resolved, many are deleted directly.

RQ3: Why do developers utilize test disabling practice? We qualitatively uncover test disabling practices and how they are used to bypass maintainability challenges. We find that most tests were disabled in the first place due to issues such as test failures, but many tests remain disabled even when the bugs are fixed. Some bugs may be marked as "Won't Fix" with the tests being disabled. We also find that developers often use disabling changes to handle other maintenance challenges in testing, such as test dependency and refactoring. 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 two groups of audiences:

(1) **Researchers**: We open an avenue for further research directions on detecting disabled tests and their relation with other aspects of software development (i.e., quality, maintainability, and performance improvements). We also highlight potential directions on assisting developers in tracking disabled tests and their co-evolution with production code.

(2) **Developers**: Our findings reveal the usage of disabled tests in many different aspects of test maintenance. Disabled tests may be used in ad-hoc ways to hide real faults or bypass test failures. Most tests are temporarily disabled until a fix is found; however, the disabled tests are not re-enabled after the fix (i.e., a bug report is closed). These findings indicate the need to assist developers with the best testing practice to trace disabled tests in practice.

Paper organization. Section 2 studies related work. Section 3 discusses our methodology and provides preliminary results. Section 4 presents our quantitative analysis results, and Section 5 presents our qualitative analysis results. Section 6 summarizes the implication of our findings. Section 7 discusses threats to validity. Section 8 concludes the paper.

2 RELATED WORK

In this section, we discuss prior studies in two areas: test maintenance and evolution, and technical debt.

2.1 Test Maintenance and Evolution

Software testing is an important practice for developing high-quality software. However, similar to source code, there may be maintenance issues related to test code. Many prior researches focus on studying various testing practices, such as test quality [2, 13, 24], maintenance [14, 28], comprehension [3], and evolution [21, 22, 31]. Bavota et al. [3] found that similar to regular code smells, test smells are prevalent in software systems and may hinder test comprehension and maintenance. Kim et al. [14] are the first to study test annotation maintenance. They uncovered test annotation smells and provided opportunities for refactoring test code using test annotations. They found that @Ignore is one of the commonly used annotations in test maintenance. Our work is different as we target test disabling/ignore practices that are not limited to annotation changes. Our work considers all types of test disabling, from commenting out tests to using @Ignore. We also propose approaches to track commented/uncommented test code with high precision.

Through a survey, Peruma et al. [19] found that developers believe using @Ignored may increase compilation time and may be harmful to code comprehension. Our work studies further why test cases are disabled at the finer-level and shows that software bug is only one of many reasons causing tests to become disabled. Zaidman et al. [31] proposed several views to mine and visualize the co-evolution of test and production code. Borle et al. [4] further study a wide spectrum of how rigorously systems on GitHub utilized test-driven development. They found that most test and production code are not updated together (i.e., the use of TDD was rare).

The most relevant prior work is the study done by Pinto et al. [21]. They conducted an empirical study on how test suites evolve (i.e., test deletion, addition, and modification) to understand the reasons for test changes. They found that in addition to test repair, most test changes are related to refactoring, deletion, and addition. In particular, they found that the key reason for test deletion is test obsolescence. Different from prior studies, our work is the first to study test maintenance from the perspective of test disabling practices. We implemented a Java annotation parser and a tool to track how disabled tests evolve. We found that test disabling changes are prevalent during test evolution, and many disabled tests remain disabled. We also manually studied the reasons for the tests to be disabled, and discussed potential challenges and future

directions on test maintenance. Moreover, Pinto et al. [21] reported 14.5% test code deletion that was once alive, while our study looks at disabled tests that are deleted (i.e., 17.1%).

2.2 Technical Debt

Cunningham [6] discussed the concept of technical debt, where short-term rewards may induce higher maintenance costs in the long run. Potdar and Shihab [23] discussed the concept of self-admitted technical debt (SATD), where developers intentionally introduced some commented-code as a form of temporary fixes. Wehaibi et al. [30] later showed that SATD induces less future defect than non-SATD; however, SATD changes are more complex to perform. Our work is the first to investigate technical debt specific to practices of disabling test code by analyzing evolution history. Tracking history has long been recognized as a crucial artifact for many empirical studies for understanding the rationale for software changes [7, 14, 22, 25]. Several approaches have been proposed to help developers and researchers better leverage source code histories. Tsantalis et al. [26] developed RefactoringMiner that can accurately track commit-level refactoring history. Grund et al. [9] implemented *CodeShovel* to accurately track method changes in evolution history. In our work on tracking disabled tests, we propose a new approach to handle newly added test code, commented-out, and uncommented-out, which was never handled in prior studies.

3 METHODOLOGY

In this section, we discuss our methodology for tracking test code evolution to identify test disabling-related changes. Tracking the evolution of program elements in commit history is an ongoing research direction, and most existing tools are not designed specifically to work on annotations and commented-out code [10, 20, 26, 27]. Therefore, we propose an approach that detects disabled tests through analyzing annotations and comments and tracks the evolution of the detected disabled tests. Our approach first detects all tests, including both disabled and active (i.e., enabled) tests in each studied version (Section 3.2), and second performs version-by-version comparison to track the evolution of the tests (Section 3.3).

3.1 Definition of Test Disabling Practice

When using testing frameworks such as JUnit or TestNG, developers can use the @Test annotation to specify a method or methods in a class as test cases. As systems evolve, some tests may become obsolete or require some changes, and developers may want to temporarily disable a test. Developers can use framework-supported annotations (i.e., @Ignore), removing the @Test annotation, or commenting out the test method/class to disable a certain test. Our goal is to study such test disabling practices in the codebase and their potential impact on test maintenance. In particular, we consider the following code changes as test disabling changes (in contrast, we consider the reverse operations as test re-enabling changes):

(1) Adding @Ignore at both the class and method level.
(2) Setting @Test(enable=false).
(3) Deleting @Test annotation.
(4) Commenting out the entire test method or class.

3.2 Detecting Disabled and Active Tests

Given one version, our approach first detects all the disabled and active (i.e., enabled) tests. Detecting active tests is straightforward: We leverages JavaParser to find all the tests with an *@Test* annotation as the testing frameworks (JUnit 4 and 5, and TestNG) of the studied systems require to have such an annotation for tests.

Detecting disabled tests requires one to design techniques per each type of test disabling practice. Each type of disabling-related change (as defined in Section 3.1) corresponds to one unique type of disabled test. The first two types, i.e., adding @Ignore and setting parameters in @Test, will result in adding explicit annotations to the disabled test methods. The third type will produce methods in test classes without an @Test(...) annotation. The fourth type will result in complete test methods embedded in comments. Similar to detecting active tests based on @Test annotation, for the first three types, we also utilize JavaParser for analyzing method annotations. For the last type, we propose an algorithm to identify commented-out test methods.

3.2.1 Analyzing Annotations for Detecting Disabled and Active Tests. For detecting active tests and some types of disabled tests, we use the off-the-shelf JavaParser [1] to extract annotations per method in test classes. The following rules are applied to decide the status of a method in test classes.

- If the annotations contain @Test (without enable=false parameter), the associated method represents an active test.
- If the annotations contain either @Test(enable=false) or @Ignore, the associated method represents a disabled test.
- If the annotations do not contain @Test, the associated method represents a *candidate* disabled test.

Note that the lack of @Test annotation is indefinite to decide a disabled test since it is common for developers to write non-test methods (e.g., helper methods) in test classes. Such *candidate* disabled tests will be further confirmed through analyzing the evolution (i.e., tracking). If a previous/later version of a candidate disabled test is an active test, this test method is confirmed disabled for the current version.

3.2.2 Detecting Disabled Tests in Comments. In addition to the disabled tests expressed by annotations, we also identify the tests that are disabled through commenting out. Algorithm 1 describes how we extract disabled tests in comments. The input *commentTarget* is either a block comment or a list of comments in consecutive lines. A *commentTarget* may contain zero or more commented-out tests. For one *commentTarget*, our tool first detects an @Test annotation (line 8) and then continues to detect a '{' in the following comment lines. The text in between could be an annotation and a method signature. Then we use a JavaParser API *parseMethodDeclaration* to confirm whether the text is indeed a method signature. In addition, we use the existence of a paired right bracket to filter out incomplete test methods that only contain method signatures.

3.3 Tracking the Evolution of Disabled Tests

Upon detecting (candidate) disabled and re-enabled tests in one version, we continue to track the evolution in the commit history, and pinpoint the related changes of such tests, e.g., one disabled test

Algorithm 1 Commented out test method detection

```
1:  Input: commentTarget
2:  Output: coTests, i.e., is a list of commented-out tests
3:  function detectCOTests(commentTarget)
4:      lines ← commentTarget.split( "\n" )
5:      coTests ← []
6:      for i←0; i<lines.length; i++ do
7:          line←lines[i]
8:          if line contains @Test then
9:              location ← location of the first occurence of '{'
10:                         in this or the following lines
11:             break if location is null
                ▷ notMethodSignature is based on a JavaParser API
12:             continue if notMethodSignature(
                                    strInBetween( i, location ) + "{}")
13:             end ← findEndOfMethodBody( lines, location)
14:             cotests.add( <the detected commented-out test> );
15:             i←end
16:         end if
17:     end for
18: end function
```

is later re-enabled. Tracking the evolution requires our approach to match program elements in every two consecutive versions.

We adapt RefactoringMiner to perform the matching because 1) RefactoringMiner is shown to have the highest precision (96.6%) and recall (94%) compared to other refactoring tools and AST diff tools [27]; and 2) RefactoringMiner can detect various types of refactoring operations, such as method/class renaming, moving and extracting methods, and modifying method signatures. Furthermore, we incorporate tracking commented-out tests in our approach. As commented-out code may be incompatible with the live code and may result in compilation errors (not supported by RefactoringMiner), we use a lightweight approach to handle the commented-out test code. In particular, a *TEST ID* (*<fully qualified class name>::<method name>(<method parameter types list>)*) is constructed based on the extracted information of commented-out tests (Section 3.2.2). For each commented-out test method, its *TEST ID* is compared with the ones in the parent and child commits for finding the paired test methods. The abovementioned points allow our approach to collect an accurate and comprehensive dataset for our study.

For each pair of matched tests in two consecutive versions, if the status of the test is modified from active (i.e., enabled) to disabled, we decide the commit is disabling-related changes. The reverse status change is determined as a test re-enabling change. If there is no matching test in the latter commit, the unmatched test is deemed deleted. If there is no matching test in the prior commit, the unmatched test is deemed newly added by the current analyzed commit, e.g., one commit may introduce commented-out tests.

Approach Evaluation. We performed an evaluation of our approach with regards to 1) detecting disabled tests in comments (Section 3.2) and 2) detecting disabled tests in commit history as these two steps may produce incorrect results.

For 1) detecting disabled tests in comments, we took all the 168 detected commented-out tests, examined them manually to decide the correctness. Our approach yields a precision of 100%. However, we cannot evaluate the recall due to the lack of oracles. For 2), as tracking is performed at commit level (e.g., whether one commit is disabling-related), we used stratified sampling to take a statistically significant (95%±5%) sample of 584 cases on three types of changes, namely disabling a test (364), re-enabling a test (122), and deleting a disabled test (98). Then we manually examined the correctness of each sampled commit. Table 1 shows the precision of the three types of changes. Our approach achieves a precision of 97% for all the sampled cases and a precision of 98%, 96%, and 92% for the three change types, respectively.

Table 1: Precision of test tracking, i.e., detecting the three types of commit changes, including disabling a test, re-enabling a test, and deleting a disabled test.

	Disabling a test	Re-enabling a test	Deleting an disabled test	Total
Sample	364	122	98	584
Precision	98%	96%	92%	97%

Upon manual examination, we identify the following sources of false positives. First, due to framework migrations, developers may need to add or delete annotations. For example, migrating from TestNG to JUnit4 will remove the @Test on the class, which will be detected as ignoring a test class, and adding @Test on each method will be detected as unignoring test methods. Second, due to the limitations of RefactoringMiner, duplicating one file to two similar files and merging two files into a single file cannot be detected. For example, in Apache Camel (*9da3f5af*), the *Web3jConsumerIntegrationTest* is duplicated to *Web3jConsumerTransactionsTest* and *Web3jConsumerLogTest*. However, RefactoringMiner only reports *Web3jConsumerIntegrationTest* is renamed to *Web3jConsumerTransactionsTest*. Thus, we detect *Web3jConsumerLogTest* as a newly added class. In Apache Flink (*8d3a74f9*), *StatefulJobSavepointFrom12MigrationITCase* and *StatefulJobSavepointFrom13MigrationITCase* are merged to *StatefulJobSavepointMigrationITCase*, but RefactoringMiner only reports *StatefulJobSavepointFrom12MigrationITCase* is renamed to *StatefulJobSavepointMigrationITCase*. Thus, we detect *StatefulJobSavepointFrom13MigrationITCase* as a deleted class. Lastly, RefactoringMiner does not work for commented-out tests, so renaming a commented-out test will be detected as deleting a commented-out test and adding a new commented-out test.

3.4 Studied Systems

Table 2 shows an overview of the studied systems. To obtain high-quality repositories to make our results more reliable, we select the studied systems by following three selection criteria. First, we selected the top 1,000 Java systems on GitHub ordered by popularity (i.e., stargazer count). We also ensured that the repositories are not forks as they may not be part of the main branch and not actively maintained. Although it would be interesting to study disabled tests from other branches that consist of feature additions or bug fixing activities, as they may involve more frequent usages of test disabling, our research only considers disabled tests that are merged into the main branch, as they may indicate more challenging maintainability tasks that could not be resolved at the time.

Table 2: An overview of the studied systems (from 2015 to 2020).

	Test LOC		Source LOC		
Systems	2015	2020	2015	2020	Total # Commits
Camel	355K	533K	289K	607M	22,584
Cassandra	28K	78K	177K	216K	4,336
Cloudstack	51K	80K	1M	519K	4,698
Druid	22K	147K	64K	242K	4,181
Flink	82K	371K	105K	407K	13,997
Hadoop	349K	660K	463K	810K	13,668
Hbase	168K	287K	433K	409K	7,271
Hive	104K	269K	524K	1M	8,221
Ignite	162K	500K	253K	577K	15,397
Incubator-pinot	1.6K	75K	11K	196K	6,179
Kafka	2K	133K	11K	132K	5,845
Maven	14K	17K	44K	48K	660
Openfire	1.2K	5.2K	179K	94K	2,097
Orientdb	74K	188K	140K	360K	8,452
Storm	2.7K	35K	61K	240K	4,554
Total	1.4M	3.3M	3.8M	5.9M	122K

Second, we discarded the systems that are below the 90th percent quantile in terms of size (i.e., lines of code), repository popularity (i.e., stars), and the number of commits collectively. Namely, we only study repositories that fall inside the top 10% in all of the mentioned criteria. Finally, we discarded inactive repositories that did not have any commits in 2020. We ended up with 15 systems, i.e., Camel, Cassandra, Cloudstack, Druid, Flink, Hadoop, Hbase, Hive, Ignite, Incubator-Pinot, Kafka, Maven, Openfire, Orientdb, Storm. We analyze the code changes in these systems from January 2015 to January 2020. These studied systems cover different domains, ranging from distributed databases, stream processing frameworks, message brokers, and group chat servers.

4 A QUANTITATIVE STUDY ON THE TEST DISABLING PRACTICE

In this section, we quantitatively analyze the prevalence of the test disabling practice. We also study the time it takes for developers to re-enable a test and the evolution patterns of disabled tests.

RQ1: How Common are Test Disabling Changes?

Motivation. Many prior studies focus on studying maintenance challenges caused by technical debt [6, 16, 20, 25]. However, there is less empirical evidence on the technical debt in the test code thus far, especially on technical debt related to disabling practices. Developers may disable tests as code evolves, which may cause future maintenance challenges. As a stepping stone to understanding test maintenance challenges, in this RQ, we study the frequency of test disabling practices.

Approach. We study how frequently developers disable a test at both class and method levels. Disabling tests at the class level would prevent executing all test cases within the class, whereas disabling at the method level would stop executing a single test case (e.g., a method with an @Test annotation). To provide a comparative statistic, we show the prevalence of test disabling changes (i.e., disable/re-enable/delete) along with common test code transformations at the same program element level by following prior studies [12, 14]. Specifically, we compare test disabling changes at method level with Rename Method, Rename Parameter, and Change Parameter Type, and at class level with Rename Class. We use a tool, called RefactoringMiner, implemented by Tsantalis et al. [26] to detect rename and type changes. Tsantalis et al. [26] reported that RefactoringMiner could detect refactoring activities with an average precision of over 99% and recall of over 93%. Despite differences in the two practices, such comparison is reasonable because these common code refactorings occur at the same program level.

Table 3: Frequency comparison between test disabling changes and the common test code refactorings at the same program element level.

	Method Level		Class Level		Total	Total
	Total	Per Commit	Total	Per Commit	Changes	Commits
Test Disabling Changes	2,581	2.7	530	0.6	3,111	949
Refactoring	2,495	1.9	120	0.1	2,615	1,334
Δ % Percentage	+3.4%	+42%	+341%	+500%	+19%	-29%

Table 4: The frequency of various types of test disabling-related changes. *Disabling Tests* shows the total number of changes that disable tests. *Re-enabled Tests* shows the total number of changes that re-enable tests and whether developers simultaneously modified the tests (i.e., modified vs. unmodified). *Deleting Disabled Tests* shows the number of changes that delete disabled tests.

	Disabling	Re-enabling Tests		Deleting
	Tests	Modified	Unmodified	Disabled Tests
Method	2,581	314	486	762
Class	530	115	96	87
Total	3,111 (62.5%)	429 (8.6%)	582 (11.7%)	849 (17.1%)

Results. ***Test disabling changes is prevalent during test maintenance and has a similar change frequency compared to test refactorings.*** Table 3 compares the prevalence of the disabling changes with that of the refactoring changes in test code. As shown in Table 3, at the method level, the number of test-disabling changes is comparable to the number of test refactorings (i.e., +3.4% difference), and at the class level, the test-disabling changes are performed more frequently than the rename class refactorings (i.e., +341% differences). We also observe that the total test-disabling changing commits are less than the total test refactoring commits. Despite this, at both method and class level, the average test-disabling changes per commit are higher than that of test refactorings (i.e., +19% difference). Based on these results, we find that test disabling changes are prevalent in practice and are comparable to traditional refactorings.

Table 4 presents the frequency of three types of test disabling-related changes: disabling a test, re-enabling a test, or deleting a disabled test. We find that 62.5% (i.e., 3,111/4,971) of the disabling-related changes disable the test, which is the most frequent change among all the change types. 20.3% (i.e., 1,011/4,971) of the disabling-related changes re-enable the test. Moreover, a non-trivial percentage (i.e., 42%) of the re-enabling changes modify the test. In other words, many of the disabled tests remain the same when they are re-enabled. We also find that a significant number (17.1%) of the test disabling changes delete disabled tests. In the next RQ, we further study the destination of each disabled test and its change pattern.

Table 5: Change patterns of disabled tests. Unresolved tests represent the tests that remain disabled at the end of the studied period. Resolved tests represent the tests the are either re-enabled or deleted completely at the end of the studied period.

Change Pattern	Frequency
Change patterns for unresolved tests	
DISABLED	1,229
DISABLED → RE-ENABLED → DISABLED	21
DISABLED → DELETED → DISABLED	1
Total	*1,251*
Change patterns for resolved tests	
DISABLED → RE-ENABLED	871
DISABLED → DELETED	824
DISABLED → RE-ENABLED → DISABLED → RE-ENABLED	46
DISABLED → RE-ENABLED → DISABLED → DELETED	24
DISABLED → RE-ENABLED → DISABLED → RE-ENABLED → DISABLED → RE-ENABLED	1
Total	*1,766*

Developers frequently disable tests in software development, and the frequency of such practice is comparable to common refactorings at the same program element level. We also find that many disabled tests may be re-enabled without any changes or may be deleted directly.

RQ2: What is the Change Pattern of Disabled Tests?

Motivation. As shown in RQ1, test disabling practice has a non-negligible presence during test maintenance and evolution. In this RQ, we further study the evolution patterns of disabled tests, their destination (e.g., finally re-enabled or stay disabled), and how long a test remains disabled. Studying the evolution of disabled tests may quantitatively show the process of how developers maintain disabled tests, whether the corresponding issues are fixed immediately or persist for a long time, and whether the corresponding issues are improperly fixed and cause the tests to become disabled again in the future.

Approach. Our goal is to study the life-cycle of every disabled test. In RQ1, we analyze the test disabling changes by studying their frequency. However, bugs that are hard to fix may cause multiple rounds of changes to the test code. Hence, we carefully track the evolution history by utilizing the full commit-level history of the disabled test to report their evolution pattern and final destination. To study the change pattern more accurately, we trace refactoring activities (e.g., rename) performed on disabled tests, using RefactoringMiner [26]. Additionally, we study the longevity of disabled tests by mining the number of days between the changes that disable the tests and the changes that either re-enable or delete the disabled tests.

Results. ***Many disabled tests (41%) remain disabled in the studied period. For the resolved disabled tests, 47% were deleted in the codebase.*** Table 5 shows the evolution patterns of disabled tests, which highlights how disabled tests evolve ever since they were born. Note that since a test may undergo multiple rounds of changes in the studied period, the total frequency of evolution patterns should be lower than RQ1 where we report the total raw frequency. We also categorize the statistics into unresolved and resolved cases to indicate the final destination of disabled tests. Unresolved tests refer to the tests that stay disabled, and resolved tests refer to the ones either being re-enabled or deleted. We find that most disabled tests (41%, 1,251/3,017) stay unresolved as the destination. For the resolved cases, 28% (848/3,017) become deleted, and 30% (918/3,017) become re-enabled. There are 21 cases where developers tried to resolve a disabled test but eventually changed them back to being disabled. After some investigation, we find that these tests are disabled again due to three main reasons. First, developers revert the commit due to a mistake. Second, developers re-disable the test code later when the test fails again. Third, developers re-disable the test code whenever there is a version update of one external dependency. There are also 24 cases where developers tried to resolve the ignored test but eventually deleted them. We observe that these tests were disabled for a long time and may have become obsolete as developers suggest deleting the tests instead of updating them.

Table 6: The distributions of the average time (in days) for a disabled test to become re-enabled, deleted, or remain disabled (i.e., developers did not modify the test in the studied period after it was re-enabled).

	Time (in days)					
	Min.	25%	50%	75%	Max.	Mean
Re-enabled	8.4	17.1	39.4	65.9	431.2	61.8
Deleted	4.8	19.9	92.3	222.1	696.6	158.7
Remain Disabled	142.0	508.0	793.2	1096.4	1475.7	797.4

Table 6 shows the distributions of the average time (in days) it takes for a disabled test to become re-enabled or deleted (i.e., resolved). We also show similar statistics for the tests that stay disabled (i.e., unresolved). We observe that the median time for developers to re-enable a test is 39 days. However, it often takes over three months (median is 92 days) for developers to delete a disabled test. In general, there is a higher possibility that a disabled test may become deleted if it has been disabled for a more extended period. One likely explanation is that many of the disabled tests may have become obsolete. We notice similar patterns of obsoleteness across all types of disabled tests.

Finally, for the tests that remain disabled, most of them have been disabled for several years. It is likely that these disabled tests are "forgotten" by developers and remain in the codebase. In RQ3, we manually study the reasons for the tests to be disabled and re-enabled.

Overall, it takes a longer time for the disabled tests to be deleted (median time is three months) than to be re-enabled (median time is 39 days). Many tests that remain disabled have been disabled for years.

5 A QUALITATIVE STUDY ON DISABLED TESTS

RQ3: Why do developers utilize test disabling practice?

Motivation. As previous RQs reveal, disabling tests is a ubiquitous practice during software evolution. The test disabling practice is a

double-edged sword. On the one hand, it provides developers with convenience in bypassing test-related issues. On the other hand, it may be used to bypass some maintenance difficulties, which can result in a silent and long waiting period for the tests to be re-enabled, if ever. Test disabling mechanism may hinder software reliability as the disabled tests may remain disabled indefinitely in codebases. In particular, there is a lack of tools to manage the life cycle of disabled tests and assist developers in proactively re-enabling the temporarily disabled tests. In this RQ, we perform a qualitative study to understand why developers utilize test disabling practice, i.e., the scenarios that developers utilize such convenience of disabled tests. Categorizing the scenarios will reveal the common challenges developers may face in maintaining disabled tests and test maintenance in general. Obtaining such understanding on disabling tests will inspire future tools that can better manage disabled tests for improving quality assurance activities.

Approach. To understand the motivations of utilizing test disabling practice, we analyzed and categorized a statistically significant sample of disabled test instances. We combined the disabled tests from all the studied systems and adopted the stratified sampling technique to sample each studied system independently for producing a statistically significant sample (95±5%). We also found 9 incorrectly detected instances by our tool, i.e., a 2.6% false positive rate, and excluded them in Table 7.

Our manual study involves two phases:

Phase I. The first two authors of the paper (A1 and A2) independently derived an initial categorization by manually inspecting the relevant software artifacts such as commit messages, test code, comments surrounding the test code, and bug reports if available. Additionally, we use *git log* to check out other relevant commits on the same set of modified source code files to gain supplementary insights if the current commit lacks sufficient information.

Phase II. A1 and A2 unified the derived reasons and compared the assigned reason for each evolution pattern. Any disagreement was discussed until reaching a consensus. The inter-rater agreement of the coding process has a Cohen's kappa of 0.7, indicating a substantial level of agreement [5]. To encourage the replication of our results, we have made the dataset available[1].

Results. Table 7 shows our manually derived taxonomy of the reasons developers disable the tests. Below, we discuss each category in detail.

- **Hiding Test Failure (40%).** Developers frequently disable some tests when test failures occur. The most common cause (81/131) is that bugs are introduced during software maintenance. While working on fixing the bugs, developers may temporarily disable the failing test cases, especially when the bugs require non-trivial effort and time to fix. However, only 32/131 of the disabled tests are re-enabled after the relevant bugs are fixed. In this case, we also notice that developers do pay a certain effort to provide some traceability of the disabled tests, such as creating bug reports on disabled test cases as a reminder to re-enable such tests. In contrast, some issues are difficult to fix, and test failures may have persisted for a non-trivial time. Developers adopt test disable to remove test failures explicitly without working on fixing the bugs. *7/131 of the tests were disabled due to failure and were never re-enabled in our studied period.* For such cases, test disabling practice is used by developers as a convenient way of bypassing test failures while keeping the test code in the repository. However, we found that there is no traceability provided for developers to track these disabled tests. We could not find any mention of the disabled tests in Jira issues or in documents. These disabled tests may remain forgotten with the issues remain unfixed.

  Another main motivation (42/131) is to avoid test failures caused by flaky tests. As flaky test results are nondeterministic, diagnosis can be challenging. After developers fix the issues (i.e., flaky tests no longer fail), they may re-enable the tests. In our studied cases, *only 7/42 flaky tests are re-enabled later*, aligning with the study by [15], who reported that over half of flaky tests remain unfixed. We also uncovered two test failures due to slow tests, for which developers simply disable the slow tests without either fixing the test or the source code. Finally, one failure is caused by library incompatibility in Travis CI. In *Openfire (ddb20ffe)*, developers discuss that @Parameterized is not supported by the Ant build system installed in Travis CI, causing failure in tests where @Parameterized is used. Developers disable the affected tests as a convenient solution while waiting for the incompatibility to be resolved by framework developers.

> In total, only 31% of the tests are re-enabled after the bug fix. The remaining disabled tests are either disabled indefinitely (i.e., due to lack of solution) or deleted. We also find that, for such permanently disabled tests, there is often a lack of traceability in GitHub or Jira issues.

- **Precautions during Feature Maintenance (23%).** Developers commonly utilize test disable practice to avoid potential test failures during maintenance activities, such as adding new features and refactoring. For 62/78 cases, we find that developers precariously disable the test cases that may fail in the process of feature implementation due to incomplete functionalities (e.g., the feature takes more than one commit to finish). For example, in *Flink (df448625)*, developers commented out the tests related to retrieving log files and left a comment saying that *"TODO activate this test after logging retrieval has been added to the new web frontend"*. We also notice that developers may indicate relevant bug issue IDs when disabling test cases. For example, in *Hadoop (18fe65d75)*, developers left a comment as *"requires HDDS-801. Requires once feature is in place"*. Although we find that developers may try to add traceability on disabled tests, the current practice remains ad-hoc (i.e., by using only comments in the code). Developers may use disabled tests to harbor some temporary tests and delete such tests after the feature is complete, i.e., being replaced by an official test (e.g., *Orientdb (0fc9bee1). Nevertheless, among the 62 cases we examined in this category, only 30 tests are re-enabled, and the others remain disabled in the studied systems.*

  In addition to new features, we find that developers may disable the tests of deprecating features (instead of removing them). Developers may choose to disable the tests temporarily while replacing deprecated features and adopt the disabled tests once the new implementation is finished. However, we observe there are cases where the tests for the deprecated features remain disabled

[1] https://github.com/boyang9602/FSE_Ignore_Test

Table 7: Qualitative analysis result: a taxonomy of why developers disable tests.

Categories	Motivation	#Frequency
- Hiding Test Failures		**133 (40.6%)**
Disabling tests while working on bug fixes	Developers temporarily disable failing tests while working on fixing the bugs.	81
Flaky test	Developers may disable flaky tests to avoid occasional failures.	42
Won't-fix bugs in code/test	Developers disable tests to avoid failures as they will not work on fixing the failures due to difficulty.	7
Slow test	Test failures due to long running time, so developers disable such tests to avoid failures.	2
Library incompatibility	JUnit annotation (@Parameterized) was not supported by Ant in the used CI platform. Thus, the test was disabled.	1
- Precautions During Feature Maintenance		**78 (23.8%)**
New/Improved Features	In the process of implementing new or improving existing features, developers may temporarily disable relevant tests or introduce new tests that are disabled (e.g., commented-out tests) to avoid potential failures.	62
Deprecation	Developers may disable relevant tests in the process of deprecating features.	12
Refactoring	Developers may disable tests during refactoring.	4
- Diverting to Manual Testing		**38 (11.6%)**
Require manual input	Developers need to manually run tests that need to be manually configured (e.g., database setup and secret keys).	24
Expensive Test	Developers need to manually run tests that are expensive to run and may not have to be run all the time (e.g., performance and migration test).	13
Experimentation	Developers manually run tests that are under experimentation.	1
- Dependency Issue		**21 (6.4%)**
Difficulties in maintaining external dependencies	External dependency is hard to maintain due to various reasons. Developers may need to disable tests when there are bugs in the external dependency, unexpected version changes, or dependencies are hard to integrate or use.	17
Waiting for functionality update in external dependency	Developers disable tests while waiting for feature improvement in the external dependencies.	4
- Test Design Issues		**16 (4.9%)**
Selective test inheritance	Developers disable tests to disable unneeded inherited tests, while selectively reusing some inherited tests.	14
Redundant Test	Developers disable tests that are redundant and covered by a different test class.	2
- Other Reasons		**41 (12.5%)**
Unknown	Lacks of explicit mention of why tests are disabled (e.g., no relevant Jira issues and comments).	32
Obsoleteness	Developers disable obsolete tests.	5
disable by mistake	Developers accidentally disable the test.	4

in the codebase. Lastly, developers sometimes disable tests during refactoring (4/78). For example, in *Openfire-97f7cf3f* and in TrustStoreConfigTest.java, developers commented out the entire class prior to extracting OpenfireX509ExtendedTrustManagerTest and removing few code duplications using a helper class, KeystoreTestUtils.java.

> Developers may temporarily disable tests during feature maintenance or refactoring. However, we find that 50% of such disabled tests remain disabled.

- Diverting to Manual Testing (11.2%). Developers may disable some tests with the plan to manually running them. For the studied cases in this category, test reconfigurability was the most common problem causing developers to disable the test (24/38). For example, we find that some tests are disabled for manual testing due to the need to manually configure the access key and secure key (e.g., *Camel (ba22a8175f94a)*) or setup databases (e.g., *CloudStack (96c38bf4)*). Such cases require manual testing as the tests depend on resources that must be manually started or configured prior to the test execution. Another 13/38 of the tests in this category are disabled because they are expensive to run (e.g., migration or performance tests). For example, as discussed in *Flink (b7ae3e5338): ManualWindowSpeedITCase*), *"When doing a release, we should manually run theses tests on the version that is to be released and on an older version to see if there are performance regressions."* However, it is difficult to know if developers remember to manually run these tests before each release, and disabling these tests may result in higher maintenance costs (e.g., only discovering issues before the release). Finally, we find one rare case in *Camel (bd1661b248): JavaSocketTests*, where the developer introduces a commented-out test code as it is part of experimental code.

> Developers manually test expensive resources that should only be executed under particular circumstances, such as migration or performance testing.

Dependency Issue (6.2%). Without good mocking strategies, we find that external dependencies may become hard to maintain (17/21) and a source of technical debt, causing tests to fail and

be disabled. For example, in *Camel (35b83b1d)*, we find that bugs in the external dependencies cause test failures (i.e., *"Upgrade smack due to bug in smack 4.0.6"*). In another case, in *Camel (40ae73c4)*, due to unexpected version changes in external dependencies, the test fails and is disabled (i.e., *"It looks like the problem is embedded XMPP server. It was overridden to 2.21 currently"*). In total, only 5/17 cases were re-enabled after the issue was fixed. Interestingly, as shown in *IGNITE-9920*, the test continues to be disabled even if the bug report is reported closed with a *WONT'T FIX* resolution due to difficulties in test maintenance. The other tests remain disabled due to similar reasons.

Developers may also disable tests while waiting for new features to be added in external dependencies. For example, in *Druid (4b3bd8bd)*, developers comment out several tests as they need to wait for an external dependency (i.e., Joda) to release a new feature. However, we find that only 1/4 cases become re-enabled later.

> Developers may disable tests due to issues/updates in external dependencies. However, 71% of the disabled tests in this category remain disabled or are deleted due to reasons such as test maintenance difficulties.

Test Design Issues (5%). Developers may disable tests when changing/improving test design. We find that developers may leverage inheritance to reuse part of the tests in parent classes. However, developers may disable some of the inherited yet unneeded tests in child classes. A reverse may also happen where developers re-enabled the disabled tests in the child classes (*Ignite (63b9e1653d)*). Our finding shows that there may be maintenance or designing challenges when handling test inheritance. Therefore, developers need to decide whether an inherited test should be executed or not. Additionally, there are cases where developers find the same tests that exist in multiple test classes and decide to disable the redundant tests.

> Developers may disable tests to bypass test design limitations related to test inheritance.

- Other Reasons (12.5%). We categorize the remaining disabled tests that do not belong to any of the above-mentioned categories as *"Other Reasons"*. We find that there are 32 cases of disabled tests where we cannot find any discussion/comment. We categorize these cases as unknown. For example, developers may only include a commit message, such as *"Disable Test"*, without any other explanation or reference to other software artifacts (e.g., Jira), where only 7 out of 32 were eventually re-enabled. We find 5 cases where developers disable the tests because they become obsolete. Finally, there are 4 cases where developers disable the test accidentally (e.g., commented out the test) and thus were re-enabled immediately afterward.

> Developers did not provide the reasons nor traceability (e.g., Jira issues) for many (32/338) of the disabled tests. Most of such disabled tests remain disabled in the studied systems. We also find cases where developers disabled obsolete tests, and they may also disable some tests by mistake.

6 DISCUSSION AND IMPLICATION

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

6.1 Discussion and Implication for Researchers

R1: Developers use disabled tests to bypass test failures, which may affect test maintenance and code quality. Future studies should investigate its impact on software quality.

As we find in RQ2, tests may be hard to fix immediately and may remain disabled for an extended period. Some re-enabled tests are again disabled later due to inadequacy of the bug fixes as bug fixing tasks are difficult. Moreover, as found in RQ3, developers may disable tests to temporarily hide test failures, such as disabling flaky tests, while the bugs remain unfixed. Although these tests are necessary for revealing faults, they may no longer guard the software against regressions and uncover the possible presence of new bugs. Therefore, future research must study the impact of disabled tests on software quality from two aspects. First, an interesting direction is to study the relationship between disabling tests and software defect proneness to provide additional insights into the impact of disabled tests on software quality. Second, future studies may quantitatively assess the impact of disabled tests on fault detection capabilities using mutation analysis [11].

R2: There is a lack of automated support on tracking disabled tests, which may lead to "forgotten" tests. Future studies may consider providing traceability support to developers.

During our manual study, we notice that many disabled tests are not referenced in issue reports and are not tracked by any software artifact. For example, developers may commit test disabling changes with only mentioning in the commit message that a test is disabled. In some cases, we find that the bug may be resolved, but the test is still disabled. Due to the lack of traceability and documentation, most of these disabled tests may then be "*forgotten*", and stayed disabled for several years and are never re-enabled. Future research may consider providing automated traceability to track the disabled tests and better assist developers during test evolution.

R3: Developers may disable some tests and divert them to manual testing. Future studies should investigate approaches to provide better automation support.

As we find in RQ3, 11% of the tests are disabled because they are difficult to run automatically or are time-consuming. However, delaying test execution may result in accumulated maintenance overhead (e.g., bugs are only revealed at the late stage of the development or before the release). Since these tests need to be manually executed, it is also possible that developers may forget to run them. Future studies may also investigate better mocking approaches and adopt test reduction or prioritization [8, 18] to ensure these tests can still be included in part of the continuous integration process while maintaining acceptable test overhead.

R4: Future studies should investigate the impact of test obsolescence and the co-evolution between test and source code.

In our study, we find that developers use test disabling for a wide range of maintainability tasks, yet only a few were ever re-enabled in practice. Many of the disabled tests are deleted as they become

Figure 1: An example of an invalid block comment that cannot be detected by the tool. Invalid comment represents a mix between natural language and code.

Original Method

```
@Test
/* this is a demo*/
public void demo() {
}
```

Commented using Block Comment

```
/*@Test
this is a demo
public void demo() {
}*/
```

obsolete. As systems evolve, some tests may become outdated and may need to be updated. However, it is not clear to which degree do developers maintain tests to keep up with the development, and whether there are replacement tests for the tests that were deleted. Future studies on test obsolescence and their co-evolution with source code may provide better support to developers on test maintenance.

6.2 Discussion and Implication for Developers

D1: Assisting developers with best testing practices about how to maintain disabled tests.

As found in RQ1 and RQ2, test disabling is prevalent in test maintenance, but most disabled tests remain disabled. Developers often disable tests due to a bug; however, they rarely open a new Jira issue to track the process of re-enabling the test code. For example, out of the 141 samples that remain disabled (RQ3), only 10% are related to unresolved issues and 22% of the tests that remain disabled do not have any documentation on Jira. For the remaining 68% of the disabled tests, they remain disabled even after the issues were fixed. As an example, in *Hive (22df53b6)*, several tests remain disabled and forgotten even after their bug issues were closed (e.g., HIVE-18341). To better trace these disabled tests, developers should consider adding Jira issues, or similar artifacts, to document tests that require an update. Otherwise, the non-traceability of disabled test code may lead to worse software quality in the long term.

7 THREATS TO VALIDITY

In this section, we discuss threats to validity of our study.

7.1 Internal Validity

Since commented code may exist in countless different formats, it may be impossible to find a generalized rule to detect all the cases. Figure 1 shows one such example, where there is a mix between the commented-out code and natural language. Therefore, our tool may not be able to detect all the commented-out methods. Nevertheless, to minimize the threat, we treat the consecutive line comments as a single target and detect each target as multiple commented-out methods. We apply the same rule when there could be blank lines between the different parts of a commented-out method. Namely, we allow our rules to be tolerant for a single line of code, but we treat it as two different targets for a number higher than one. Despite this issue, the precision of our tool for test status is high (98%), where in fact there are no false positives in terms of commented out tests. Our approach also relies on RefactoringMiner to detect and track refactoring changes. As shown by Tsantalis et al. [26], RefactoringMiner has high precision and recall, which should not affect our results. It is possible that some commented-out code is only meant as a template code for future implementation and could be non-compilable. Our tool does not check for these instances, and they may exist in our study and skew our understanding of the manual study. Moreover, we only consider @Test annotation to determine the test method and test class. It is also possible that files that use an older version of Junit (i.e., Junit3) may not use @Test annotation to indicate a test method or class. Our tool does not check for these instances and may miss some disabled tests. Finally, we do not consider partially commented out tests, such as commenting out assertions. Compared to disabling a test completely, partially commented out tests may involve more complex change, such as when tests contain multiple assertions in a test.

7.2 External Validity

Our studied systems are all open source implemented in Java, so the result may not be generalized 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. Future studies are encouraged to replicate our experiment on other systems and systems implemented in different programming languages.

7.3 Construct Validity

In RQ3, we conduct a manual study on the reasons that the tests become disabled. We conduct the study on a statistically significant sample using a 95% confidence level and 5% confidence interval. To reduce the biases in our manual study result, two of the authors independently studied the sample and compared the results. Any discrepancy is discussed until a consensus is reached. We computed the Cohen's Kappa, and found that the level of agreement is substantial between the two authors (0.7).

8 CONCLUSION

Similar to source code, there are bugs and maintenance challenges in test code. As a result, developers may bypass a test failure by disabling the test, i.e., adding @Ignore or comment out the test. Such tests disabling practices, when misused, may cause technical debt and harm the long-term maintenance. In this paper, we conduct the very first empirical study on test disabling practice in Java systems. We first implement a tool to detect and track the changes of disabled tests in software development history. Then, we conduct both quantitative and qualitative studies on test disabling changes. We find that: 1) Test disabling is a prevalent practice in test evolution and has a similar frequency level compared to test refactoring at the same program element level. 2) Most disabled tests remain disabled and have been disabled for years. Many of the disabled tests are either re-enabled without any code change or deleted directly. 3) Our manual study highlights the reasons for the tests to be disabled. We find that most tests are disabled due to maintenance challenges (e.g., flaky tests and required to do manual testing) rather than waiting for bug fixes. Moreover, most disabled tests remain disabled even after the bugs are fixed. Our discussions provide possible future research directions to further improve test maintenance and suggestions to developers to better track disabled tests so that they are not "forgotten".

REFERENCES

[1] [n.d.]. *JavaParser*. Retrieved Feb, 2021 from https://javaparser.org/

[2] Dimitrios Athanasiou, Ariadi Nugroho, Joost Visser, and Andy Zaidman. 2014. Test Code Quality and Its Relation to Issue Handling Performance. *IEEE Trans. Software Eng.* 40, 11 (2014), 1100–1125. https://doi.org/10.1109/TSE.2014.2342227

[3] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and David W. Binkley. 2012. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *28th IEEE International Conference on Software Maintenance, ICSM 2012, Trento, Italy, September 23-28, 2012*. IEEE Computer Society, 56–65. https://doi.org/10.1109/ICSM.2012.6405253

[4] Neil C. Borle, Meysam Feghhi, Eleni Stroulia, Russell Greiner, and Abram Hindle. 2018. Analyzing the effects of test driven development in GitHub. *Empir. Softw. Eng.* 23, 4 (2018), 1931–1958. https://doi.org/10.1007/s10664-017-9576-3

[5] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46.

[6] Ward Cunningham. 1993. The WyCash portfolio management system. *OOPS Messenger* 4, 2 (1993), 29–30. https://doi.org/10.1145/157710.157715

[7] Beat Fluri, Michael Würsch, and Harald C. Gall. 2007. Do Code and Comments Co-Evolve? On the Relation between Source Code and Comment Changes. In *14th Working Conference on Reverse Engineering (WCRE 2007), 28-31 October 2007, Vancouver, BC, Canada*. IEEE Computer Society, 70–79. https://doi.org/10.1109/WCRE.2007.21

[8] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical Regression Test Selection with Dynamic File Dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (Baltimore, MD, USA) *(ISSTA 2015)*. Association for Computing Machinery, New York, NY, USA, 211âĂŞ222. https://doi.org/10.1145/2771783.2771784

[9] Felix Grund, Shaiful Chowdhury, Nick C Bradley, Braxton Hall, and Reid Holmes. [n.d.]. CodeShovel: Constructing Method-Level Source Code Histories. ([n. d.]).

[10] Felix Grund, Shaiful Alam Chowdhury, Nick Bradley, Braxton Hall, and Reid Holmes. 2021. CodeShovel: Constructing Method-Level Source Code Histories. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE 2021)*. 13.

[11] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (2011), 649–678. https://doi.org/10.1109/TSE.2010.62

[12] Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. 2020. *Understanding Type Changes in Java*. 629âĂŞ641.

[13] Dong Jae Kim, Tse-Hsun (Peter) Chen, and Jinqiu Yang. 2021. The Secret Life of Test Smells - An Empirical Study on Test Smell Evolution and Maintenance. *Empirical Software Engineering* (2021).

[14] Dong Jae Kim, Nikolaos Tsantalis, Tse-Hsun (Peter) Chen, and Jinqiu Yang. 2021. Studying Test Annotation Maintenance in the Wild. In *Proceedings of the 43rd International Conference on Software Engineering*.

[15] Wing Lam, Kivanç Muslu, Hitesh Sajnani, and Suresh Thummalapenta. 2020. A study on the lifecycle of flaky tests. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 1471–1482. https://doi.org/10.1145/3377811.3381749

[16] Zhongxin Liu, Qiao Huang, Xin Xia, Emad Shihab, David Lo, and Shanping Li. 2018. SATD detector: a text-mining-based self-admitted technical debt detection tool. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 9–12. https://doi.org/10.1145/3183440.3183478

[17] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. 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*, Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey (Eds.). ACM, 643–653. https://doi.org/10.1145/2635868.2635920

[18] Zi Peng, Tse-Hsun Chen, and Jinqiu Yang. 2020. Revisiting Test Impact Analysis in Continuous Testing From the Perspective of Code Dependencies. *IEEE Transactions on Software Engineering* (2020), 1–1. https://doi.org/10.1109/TSE.2020.3045914

[19] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. 2019. 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, CASCON 2019, Markham, Ontario, Canada, November 4-6, 2019*, Tima Pakfetrat, Guy-Vincent Jourdan, Kostas Kontogiannis, and Robert F. Enenkel (Eds.). ACM, 193–202. https://dl.acm.org/doi/abs/10.5555/3370272.3370293

[20] Tri Minh Triet Pham and Jinqiu Yang. 2020. The Secret Life of Commented-Out Source Code. In *ICPC '20: 28th International Conference on Program Comprehension, Seoul, Republic of Korea, July 13-15, 2020*. ACM, 308–318. https://doi.org/10.1145/3387904.3389259

[21] Leandro Sales Pinto, Saurabh Sinha, and Alessandro Orso. 2012. Understanding myths and realities of test-suite evolution. In *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*, Will Tracz, Martin P. Robillard, and Tevfik Bultan (Eds.). ACM, 33. https://doi.org/10.1145/2393596.2393634

[22] Leandro Sales Pinto, Saurabh Sinha, and Alessandro Orso. 2013. TestEvol: a tool for analyzing test-suite evolution. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). IEEE Computer Society, 1303–1306. https://doi.org/10.1109/ICSE.2013.6606703

[23] Aniket Potdar and Emad Shihab. 2014. An exploratory study on self-admitted technical debt. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 91–100.

[24] Davide Spadini, Fabio Palomba, Andy Zaidman, Magiel Bruntink, and Alberto Bacchelli. 2018. On the Relation of Test Smells to Software Code Quality. In *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*. IEEE Computer Society, 1–12. https://doi.org/10.1109/ICSME.2018.00010

[25] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. 2007. /*icomment: bugs or bad comments?*/. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, Thomas C. Bressoud and M. Frans Kaashoek (Eds.). ACM, 145–158. https://doi.org/10.1145/1294261.1294276

[26] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. 2020. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering* (2020), 21. https://doi.org/10.1109/TSE.2020.3007722

[27] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and Efficient Refactoring Detection in Commit History. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. ACM, New York, NY, USA, 483–494. https://doi.org/10.1145/3180155.3180206

[28] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2016. An empirical investigation into the nature of test smells. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, 4–15. https://doi.org/10.1145/2970276.2970340

[29] Arie Van Deursen, Leon Moonen, Alex Van Den Bergh, and Gerard Kok. 2001. Refactoring test code. In *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*. Citeseer, 92–95.

[30] Sultan Wehaibi, Emad Shihab, and Latifa Guerrouj. 2016. Examining the Impact of Self-Admitted Technical Debt on Software Quality. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*. IEEE Computer Society, 179–188. https://doi.org/10.1109/SANER.2016.72

[31] Andy Zaidman, Bart Van Rompaey, Arie van Deursen, and Serge Demeyer. 2011. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empir. Softw. Eng.* 16, 3 (2011), 325–364. https://doi.org/10.1007/s10664-010-9143-7

[32] Ahmed Zerouali and Tom Mens. 2017. Analyzing the evolution of testing library usage in open source Java projects. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*, Martin Pinzger, Gabriele Bavota, and Andrian Marcus (Eds.). IEEE Computer Society, 417–421. https://doi.org/10.1109/SANER.2017.7884645