

Understanding Test Deletion in Java Applications

Suraj Bhatta
North Dakota State University
Fargo, USA
suraj.bhatta@ndsu.edu

Frank Kendemah
North Dakota State University
Fargo, USA
frank.kendemah@ndsu.edu

Ajay Kumar Jha
North Dakota State University
Fargo, USA
ajay.jha.1@ndsu.edu

Abstract—**Obsolete and redundant tests increase regression testing costs. Therefore, developers should remove them from test suites; however, identifying these tests is non-trivial. Automated techniques for identifying obsolete and redundant tests could help developers reduce regression testing costs. Nonetheless, we have limited empirical evidence of how and why developers delete tests. Therefore, in this work, we first create DELTEST, a dataset of 24,431 manually confirmed deleted tests, by analyzing 449,592 commits from seven open-source Java projects. We then perform an empirical study on DELTEST to understand test deletion. Our findings show that test deletion frequency and the number of deleted tests vary significantly across projects, suggesting that test deletion is more likely driven by project-specific needs than the broader development cycle. Developers delete only one or two tests in most commits, suggesting test deletion is mostly small and incremental. In DELTEST, 83.2% of tests are deleted along with the corresponding test classes, while 16.8% are deleted individually. We find that 91.4% of deleted tests in six projects are obsolete tests (i.e., production code is deleted), 7% are redundant tests (i.e., passing tests), and 1.6% are failing tests. The deletion of 20% of redundant tests reduces code coverage or mutation scores. We also evaluate test suite reduction (TSR) approaches on DELTEST and find that a TSR approach identifies up to 54% of the redundant tests. Our findings can help improve automated techniques for identifying obsolete and redundant tests.**

Index Terms—**Test deletion, test suite evolution, test suite reduction, test suite maintenance, regression testing**

I. INTRODUCTION

Test suite evolution is a natural process for any active software system. Changes in production code and developers' desire to improve the structural coverage of software systems are the main reasons for test suite evolution [1], [2], [3], [4]. Test suite evolution occurs through test addition, modification, and deletion, where test deletion is the least frequently used activity but occurs often [1]. Test deletion can play a critical role in reducing test suite maintenance and execution costs by removing tests that do not add value to test suites [5], [6], [7], [8], [9]. Test deletion becomes more critical in continuous integration (CI), where tests are executed frequently [10].

Existing tests may break due to changes in production code. Developers may not be able to fix some of the broken tests, resulting in obsolete tests that need to be removed [9]. It is also possible that existing tests do not break, but add no value to test suites. For example, developers may add a new test that tests code behaviors including the behavior tested by an existing test, making the existing test redundant [11], [9]. Obsolete and redundant tests increase regression testing costs without any real benefits [5], [6], [7], [8], [9]. Therefore, developers should identify and remove them from test suites. However, identifying obsolete or redundant tests is non-trivial [9], [12].

Although we can identify tests with compilation errors as obsolete tests, it is difficult to determine whether tests with assertion failures or runtime exceptions are obsolete tests [13]. They could be related to bugs introduced during changes in production code [14], [15], [7]. Identifying redundant tests is even more challenging because they do not result in errors or exceptions. Automated techniques to identify obsolete and redundant tests could help developers reduce test maintenance and execution costs. However, we have limited evidence of (1) how many tests are deleted in a project and commit, (2) how often they are deleted, (3) at what levels of granularity (e.g., test method or class) they are deleted, and (4) the reasons behind their deletion.

Pinto et al. investigated test suite evolution by analyzing test suites from 88 versions (i.e., data points) of six Java projects [1]. Although they mainly focused on test modification/repair, they also investigated test deletion. However, their study is limited to understanding reasons for test deletion. Some production and test code co-evolution studies identified types of changes in production code that resulted in test deletion, providing reasons for tests becoming obsolete [3], [16], [17]. Although the results of these studies are important to understanding practitioners' need for automated techniques, the studies have three key limitations. First, they do not perform an in-depth test deletion study. They perform limited analysis on test deletion as part of a broader test suite evolution study. Second, they do not analyze test deletion in each commit, potentially missing a lot of test deletion activities. Third, they mainly focus on why developers delete tests, leaving other important questions about how developers delete tests unanswered. No study systematically investigates both how and why developers delete tests.

To address the gap, we study test deletion by analyzing 449,592 commits from seven open-source Java projects. We first create a dataset of 24,431 manually confirmed deleted tests, DELTEST. We then examine DELTEST to understand test deletion by investigating four research questions: test deletion frequency and interval (RQ1), the number of tests deleted in projects and commits (RQ2), test deletion granularity (RQ3), and reasons for test deletion (RQ4). Test suite reduction (TSR) techniques aim to reduce test maintenance and execution costs by identifying redundant tests and removing them from test suites temporarily during regression testing [9], [8]. Therefore, we design TSR experiments and evaluate TSR approaches on DELTEST to assess their capability in identifying redundant tests (RQ5). TSR approaches have never been evaluated on developers' deleted redundant tests.

Our findings indicate that test deletion is an infrequent software development activity, accounting for only 0.47% of the studied commits. Test deletion frequency varies significantly across projects, ranging from only 14 deletions in `jfreechart` over 15 years to 1,388 deletions in `CTS` over approximately 14 years. The number of deleted tests also varies substantially across projects, ranging from 151 in `jfreechart` to 17,105 in `CTS`. This suggests that test deletion is more likely driven by project-specific needs than by the broader development cycle. Developers delete 11 tests on average in a test deletion commit. However, they delete only one or two tests in most commits, suggesting test deletion is mostly small and incremental. Developers delete 83.2% of the tests along with the corresponding test classes, while 16.8% are deleted individually. We find that 91.4% of the tests across six projects are deleted due to the deletion of the corresponding production code (i.e., obsolete), 62% of them are deleted in the same commit as the production code, while 38% are deleted in separate commits. Additionally, 1.6% of the deleted tests are failed tests, but they are all placeholder tests in a project. The remaining 7% of the deleted tests are passing tests (i.e., redundant), and the deletion of 20% of them reduces code coverage or mutation scores. Also, a TSR approach identifies 54% of the redundant tests.

To summarize, this study makes the following contributions:

- We create DELTEST, providing the first dataset of 24,431 manually confirmed deleted tests. The dataset provides the necessary foundation for training, evaluating, and comparing test deletion/reduction techniques.
- We conduct an empirical study on DELTEST to understand test deletion. Our findings provide empirical evidence necessary for building and improving test deletion techniques.
- We evaluate TSR approaches on DELTEST, assessing their capability in identifying developers’ deleted redundant tests. We discuss potential opportunities and limitations.

All our data and analysis scripts are available on our online artifact page <https://figshare.com/s/ee52fbf478ca15e1597b>.

II. CREATING DELTEST

Figure 1 shows the process of creating DELTEST. We use a combination of automated (white boxes) and manual (grey boxes) steps. We select and clone seven open-source Java projects (Step 1). We then iterate through the commits in these projects to identify candidate commits that potentially contain deleted tests (Step 2). Next, we analyze modified test files in the candidate commits to identify candidate deleted tests (Step 3). After this, we identify and remove refactored tests from the candidate deleted tests (Step 4). Finally, we manually confirm deleted tests (Step 5).

A. Step 1: Project selection

We first select six open-source Java projects used in a previous test deletion study [1]. These projects have also been used in other test suite evolution studies [3], [18], [19], [20], [16]. The projects are popular and actively maintained. Additionally, we select Android Compatibility Test

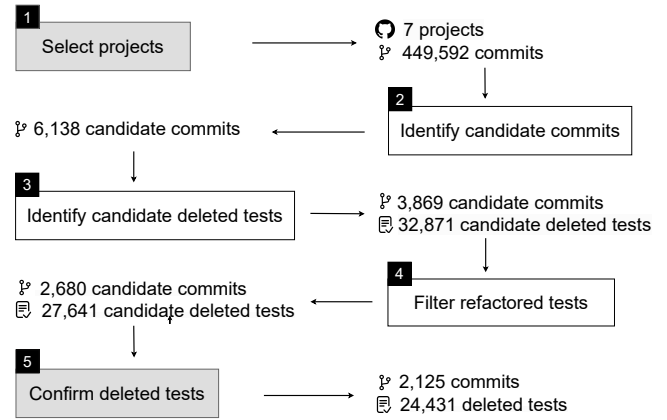


Fig. 1: The process of creating DELTEST.

Suite (CTS)¹, a free commercial-grade test suite for ensuring the compatibility of the Android framework implementations across original equipment manufacturer (OEM) partners and platform releases. We select CTS because it has a large number of tests and the Android platform is evolving rapidly². Also, although CTS is an open-source project, it is developed and maintained internally by Google, providing a different context than the other selected projects.

Overall, we select seven open-source Java projects and clone their default branch. Table I shows the selected projects with the total number of commits, which ranges from 1,771 in `gson` to 401,732 in `CTS`. In total, we consider 449,592 commits from seven projects.

B. Step 2: Identify candidate commits

We automatically analyze commits in the selected projects to find candidate commits that potentially contain deleted tests. We mark a commit as a candidate commit if all of the following criteria are true. (1) *In the default branch*. We are only interested in the commits that belong to the default/main branch. (2) *Not a merge commit*. We ignore merge commits as the changes in merge commits are already reflected in the parent branches that we analyze. (3) *Contains a modified test file with a deleted method*. We are only interested in the commits in which developers delete a method from a test file.

We use PyDriller [21] to traverse commits in the cloned projects and find commits that satisfy the above criteria. We check the first and second criteria by specifying `only_in_branch` to `default` branch and `only_no_merge` to `true`, respectively. To check whether a commit contains a modified test file with a deleted method, we first identify test files modified in the commit by checking if a filename has a `Test` prefix or suffix, which is a common naming convention [22]. We then compare methods available in the original and modified versions of each test file in the commit using PyDriller. If a method is available in the original version but not in the modified version, we identify the test file as a file containing a deleted method. The above criteria result in 6,138 candidate commits from a total of 449,592 commits that we analyze.

¹<https://source.android.com/docs/compatibility/cts>

²<https://developer.android.com/tools/releases/platforms>

TABLE I: Projects used in our study.

Project	Studied Commits				Total Test Classes	Total Tests
	Initial Commit	End Commit	Duration	Total Commits		
commons-lang	750a21e	78b4f09	20 years, 5 months	7,080	200	3,867
gson	57d1f32	1a2170b	14 years, 3 months	1,771	124	1,310
commons-math	4a8cbc2	585b04c	19 years, 7 months	7,116	406	3,173
jfreechart	6f8f85d	c77fcec	15 years, 1 month	4,219	353	2,276
joda-time	b596f23	50b0897	19 years	2,252	151	4,240
pmd	f213812	a653fb4	20 years, 6 months	25,422	747	1,577
cts	f805710	a0d9ca6	13 years, 9 months	401,732	4,542	34,004
Total	—	—	—	449,592	6,523	50,447

C. Step 3: Identify candidate deleted tests

We automatically analyze the candidate commits to identify tests that developers may have deleted. For each test file modified in the candidate commits, we parse the original and modified versions of the file using Javalang³. We then extract the methods present in the original version but not in the modified version. If an extracted method has a *test* prefix in the name or a *@Test* annotation, we consider the method as a candidate deleted test. We use Javalang for this task, because PyDriller cannot extract method annotation. However, we find that Javalang could not successfully parse some of the test files in the identified candidate commits. In such cases, we extract a *diff* of the file and then use regular expressions on the *diff* to identify candidate deleted tests. We find 32,871 candidate deleted tests in 3,869 candidate commits.

D. Step 4: Filter refactored tests

Pinto et al. found during limited manual analysis that many tests they identified as redundant and deleted were not actually deleted, they were renamed or moved [1]. To avoid this and reduce the overload of manual validation in the next step, we analyze the candidate deleted tests using RefactoringMiner [23] and filter out refactored tests. RefactoringMiner identifies 5,230 of the 32,871 candidate deleted tests as refactored tests. We filter out the identified refactored tests resulting in 27,641 candidate deleted tests in 2,680 candidate commits.

E. Step 5: Confirm deleted tests

We manually review the 27,641 candidate deleted tests in 2,680 candidate commits to confirm them. Two authors of this paper individually review each candidate deleted test, *diff* code in the commit containing the candidate deleted test, and the commit message and discuss any disagreements. We measure

the agreement rate between the two authors using Cohen’s Kappa [24]. We achieve 0.85 Cohen’s Kappa score, which is almost perfect agreement. It took approximately one month and 11 days to manually review the candidate deleted tests.

We find that 3,210 of the 27,641 candidate deleted tests are refactored tests, which RefactoringMiner fails to identify. The refactoring types in the 3,210 tests include inline method, move and inline method, move and rename method, and extract and move method [23]. We confirm 24,431 deleted tests in 2,125 commits.

III. EMPIRICAL STUDY

We analyze the deleted tests and commits that contain the deleted tests (i.e., *test deletion commits*) in DELTEST and answer the following four research questions:

- **RQ1:** *How often do developers delete tests?* This question investigates test deletion frequency and interval.
- **RQ2:** *How many tests do developers delete in projects and commits?* This question investigates the number of tests deleted in each project and commit.
- **RQ3:** *At what levels of granularity do developers delete tests?* This question investigates whether developers delete individual tests or whole test classes.
- **RQ4:** *Why do developers delete tests?* This question investigates reasons for test deletion.

We also evaluate TSR approaches on DELTEST, addressing the following research question:















- **RQ5:** *How many of the developers’ deleted redundant tests do TSR approaches identify?* This question investigates whether TSR approaches can identify developers’ deleted redundant tests.

A. RQ1: How often do developers delete tests?

1) *Method:* We first find test deletion commits in each studied project from DELTEST. We then categorize them by year

³<https://github.com/c2nes/javalang>

TABLE II: Test deletion commit (TDC) frequency and interval in the studied projects

Project	#TDC	Version with TDC /Studied version	#TDC per version		Years with TDC /Total years	#TDC per year		Interval in #commits		Interval in #days			
			Mean	Med.		Mean	Med.	Mean	Med.	Mean	Med.		
commons-lang	108 (1.53%)	16/21 (76%)		7	4	19/21 (90%)		6	4	56	32	61	23
gson	37 (2.09%)	10/19 (53%)		4	4	9/15 (60%)		4	3	47	21	139	29
commons-math	231 (3.25%)	13/14 (93%)		18	10	19/20 (95%)		12	10	32	11	31	5
jfreechart	14 (0.33%)	1/2 (50%)		8	8	4/16 (25%)		4	3	60	4	135	1
joda-time	31 (1.38%)	6/22 (27%)		5	2	9/20 (45%)		3	1	69	13	165	4
pmd	316 (1.24%)	40/64 (63%)		8	4	20/21 (95%)		16	12	83	35	23	4
cts	1,388 (0.35%)	21/21 (100%)		65	45	14/14 (100%)		99	91	291	122	3	1
All	2,125 (0.47%)	107/163 (66%)		20	5	94/127 (74%)		21	6	211	68	17	1

and project release version to understand any year/version-wise test deletion trends. We use timestamps in the commits to categorize them by year. To categorize them by release version, we select all major and minor release versions (MAJOR.MINOR) available in the GitHub repositories of the projects. We consider commits belonging to version V_n if their timestamps fall after version V_{n-1} and before version V_n release dates. Some recent commits may not have a release version yet. We exclude such commits from categorization. For CTS, we select all the official release versions⁴. To get more insights into test deletion patterns, we also calculate the interval between consecutive test deletion commits in the time interval in days and the number of commits without deleted tests. We analyze commit histories and timestamps in the test deletion commits to extract these data. We consider the time interval to be 0 days if the interval between consecutive test deletion commits is less than 24 hours.

2) *Findings*: Table II shows the results. The number of test deletion commits in the studied projects ranges from 14 in jfreechart to 1,388 in CTS, which means developers delete tests 14 and 1,388 times in jfreechart and CTS, respectively, throughout the projects' lifetime. Joda-time and gson have only 31 and 37 test deletion commits, respectively. Although commons-math and pmd have more test deletion commits than the other non-CTS projects, the numbers are not anywhere near the number of test deletion commits in CTS. We find a total of 2,125 test deletion commits in the studied projects, 65% of them are only in CTS. However, the percentage of test deletion commits in CTS (0.35%) is less than in five other projects. Only 0.47% of the studied commits across the projects are test deletion commits.

We find a total of 163 versions in the studied projects, among which 107 (66%) versions contain test deletion commits. Only CTS has test deletion commits in all the studied versions. Test deletion commits in each test-deleted version on average range from four in gson to 65 in CTS. The mean and median values for test deletion commits in each test-deleted version across the projects are 20 and 5, respectively. We also find that developers delete tests every year only in CTS. In the other projects, developers do not delete tests for one or two years in commons-lang, commons-math, and pmd, whereas they do not delete tests for six to 12 years in the remaining projects. Test deletion commits in each test-deleted year on average range from three in joda-time to 99 in CTS. The mean and median test deletion commits in each test-deleted year across the projects are 21 and 6, respectively. The graphs in Table II show the number of test deletion commits in each test-deleted version and year. The graphs do not indicate any clear year/version-wise trend in test deletion across the projects. However, test deletion occurs more frequently (i.e., significantly higher than the average) in a few of the test-deleted versions and years in the non-CTS projects. We also find that test deletion frequency in CTS has significantly increased in recent versions and years.

The number of commits between consecutive test deletion commits across the projects on average is 211, ranging from 32 in commons-math to 291 in CTS. The average number of days between consecutive test deletion commits across the projects is 17, ranging from 3 in CTS to 165 in joda-time. However, the interval data is highly skewed as indicated by the median values.

3) *Discussion*: Test deletion is an infrequent software development activity across projects, accounting for only 0.47% of the commits. However, test deletion frequency varies quite

⁴<https://source.android.com/docs/compatibility/cts/downloads>

a lot in the projects. Developers are generally reluctant to remove tests [25]. We observe this to be true for non-CTS projects, where developers often go several years without deleting tests. Most of the test deletion commits in DELTEST are from CTS, but CTS has a lower percentage of test deletion commits compared to the five other projects, suggesting that the rapid evolution of an application plays a critical role in test deletion frequency. Although CTS is an open-source project, it is developed and maintained internally by Google, and the Android platform is evolving rapidly. Future studies could further investigate how these factors influence test deletion.

The results do not show a clear version or year-wise test deletion trend across the projects, implying that test deletions are more likely driven by project-specific needs, such as the pace of project evolution, rather than by a broader development or release cycle.

RQ1: Test deletion is an infrequent software development activity, accounting for only 0.47% of the studied commits. Test deletion frequency varies significantly across projects, suggesting that it is more likely driven by project-specific needs than by a broader development cycle.

B. RQ2: How many tests do developers delete in projects and commits?

1) *Method:* We first find the number of tests developers delete in each project and test deletion commit from DELTEST. We then calculate the mean, median, and maximum number of tests deleted in a test deletion commit for each project. We also calculate the percentage of tests deleted in a test deletion commit. We get the original test suite size for a test deletion commit by identifying the total number of tests present in the direct parent commit of the test deletion commit. We then calculate the mean, median, and maximum percentage of tests deleted in a test deletion commit for each project.

2) *Findings:* Figure 2 shows the number of deleted tests in the projects. It ranges from 151 in jfreechart to 17,105 in CTS. Gson, joda-time, commons-lang, pmd, and commons-math have 260, 660, 869, 2,070, and 3,316 deleted tests, respectively. Developers delete these tests over 13 to 20 years (Table I). In total, we find 24,431 deleted tests in the projects, with CTS accounting for 70% of the tests.

Figure 3 shows the number of tests deleted in a test deletion commit. We have chopped the violin plots from the right to clearly display the mean (*) and median (.) values. Therefore, the violin plots may not represent actual maximum values. Developers delete 11 tests on average in a test deletion commit across the projects, ranging from 6 in pmd to 21 in joda-time. They delete only one or two tests in most of the commits as indicated by the median values. The percentage of tests deleted in a test deletion commit on average ranges from 0.1% in CTS to 3.4% in joda-time. However, only 1% of the tests developers delete on average in most test deletion commits as indicated by the median percentage values.

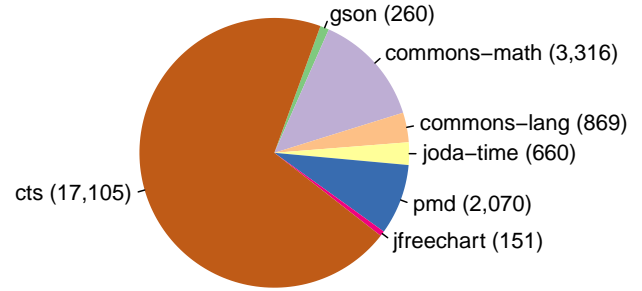


Fig. 2: Tests deleted in the studied projects

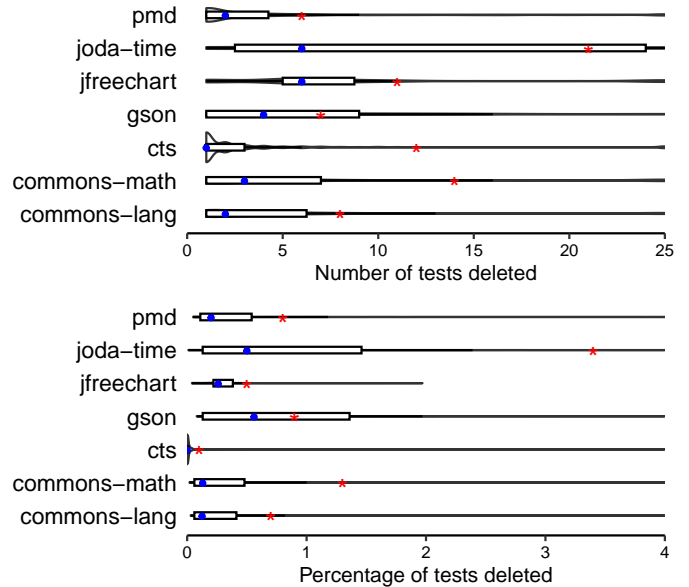


Fig. 3: Tests deleted in a test deletion commit

Developers also delete a large number of tests in a single commit reaching up to 499 (13%) in commons-math and 3,100 (18%) in CTS. Note that a test deletion commit with the maximum number of deleted tests may not have the maximum percentage of deleted tests in a project, which is the case in commons-math. It has a different test deletion commit where developers delete all the available 337 tests, reducing test suite size by 100%.

3) *Discussion:* The number of deleted tests varies significantly across projects, despite some projects having a similar total number of tests. For example, gson and pmd have 1,310 and 1,577 total tests, respectively, but 260 and 2,070 deleted tests. This again suggests that test deletion is a project-specific activity. However, the total commits in pmd is 14 times that in gson (Table I). Also, most of the deleted tests (70%) in DELTEST are from CTS, although it has the least development duration (13 years). This suggests that the pace of application evolution plays a critical role in the number of deleted tests in a project. We found a statistically significant positive correlation between the number of total commits and the number of deleted tests in a project (Spearman's correlation coefficient 0.93 and p-value 0.003), while no significant correlation (coefficient 0.04 and p-value 0.938) between the development duration and the number of deleted tests.

Deleted tests in gson are notably fewer than 400 deleted tests Pinto et al. reported in their study [1], which analyzed the project only until December 2011. We believe the main reason for this difference is their test deletion data includes tests that are renamed and moved but not actually deleted. We addressed this by filtering out refactored tests.

The median number and percentage of tests deleted in a commit across the projects are low, suggesting that most test deletions are small and incremental.

RQ2: The number of deleted tests vary significantly across projects, which is significantly correlated with the pace of application evolution. Only one or two tests are deleted in most commits, suggesting most test deletions are small and incremental.

C. RQ3: At what levels of granularity do developers delete tests?

1) *Method:* We analyze the deleted tests in DELTEST at two different granularity levels: *individual test* and *test class*. If developers delete a test along with its test class in a test deletion commit, we identify that test deletion occurs at the test class level. Otherwise, if developers delete a test without deleting its test class, we identify that test deletion occurs at the individual test level.

2) *Findings:* Table III shows the results. Developers delete 20,336 (83.2%) tests along with the corresponding test classes across the projects. These tests belong to 3,458 test classes that developers delete in 746 (35.1%) test deletion commits. Except for joda-time, developers delete the majority of tests (74.8%-86.9%) along with test classes. Only 4,095 (16.8%) tests developers delete individually across the projects. However, these deleted tests are spread across 1,464 (68.9%) test deletion commits. Note that test deletion can occur at both individual test and test class levels in a test deletion commit. That is why the sum of test deletion commits in columns third and fourth is not equal to the total test deletion commits (i.e., 2,125) in DELTEST.

3) *Discussion:* The results show that most tests become obsolete or redundant at the class level. Although only 16.8% of the tests are deleted individually, they are spread across most test deletion commits. This is the key reason we found 11 tests deleted on average in a test deletion commit and only one or two tests deleted in most commits in RQ2.

RQ3: Developers delete 83.2% tests along with test classes and the remaining 16.8% tests individually. The tests deleted along with test classes and individually are spread across 35.1% and 68.9% test deletion commits, respectively.

D. RQ4: Why do developers delete tests?

1) *Method:* We employ a mixed-method approach, combining static analysis of code changes with dynamic test execution, to identify reasons for test deletion.

TABLE III: Test deletion granularity

Project	#Tests with class	#Test class (#TDC)	#Tests without class (#TDC)
commons-lang	688 (79.2%)	70 (38)	181 (71)
gson	226 (86.9%)	43 (23)	34 (17)
commons-math	2,738 (82.6%)	339 (105)	578 (133)
jfreechart	122 (80.8%)	21 (9)	29 (7)
joda-time	291 (44.1%)	20 (7)	369 (26)
pmd	1,548 (74.8%)	512 (137)	522 (193)
cts	14,723 (86.1%)	2,453 (427)	2,382 (1,017)
Total	20,336 (83.2%)	3,458 (746)	4,095 (1,464)

Developers could delete a test because they remove the corresponding production code. Therefore, we begin by analyzing how many tests developers delete due to the removal of production code and identify them as *obsolete* tests. We first use a static analysis approach to avoid running tests on many commits. We analyze production and test code changes in each test deletion commit in DELTEST. Specifically, we identify all the production methods deleted in a test deletion commit by comparing methods in the previous and current versions of each modified file. We then consider a deleted test in the commit obsolete if any of the deleted production methods are invoked in the deleted test. The above scenario considers production and test methods deleted in the same commit. However, developers could delete them in separate commits. Therefore, we check whether invoked production methods in a deleted test are still declared in the corresponding production classes. If any of them are not declared, we consider the test obsolete. We categorize obsolete tests as *obsolete-parallel* and *obsolete-phased* if developers remove the production code in the test deletion commit and earlier commits, respectively. For obsolete-phased tests, we also calculate the interval between the test deletion commits and the corresponding production code deletion commits in days.

We next run the remaining deleted tests. For this, we check out the immediate parent commit of each test deletion commit and run the tests. We then categorize the deleted tests into the following three categories. (1) *Obsolete* if a test has a compilation error. We categorize such tests as obsolete-phased because we run them in the parent of a test deletion commit. (2) *Failed* if a test fails due to an assertion failure or runtime exception. (3) *Redundant* if a test passes. We categorize redundant tests into *confirmed redundant* if they do not reduce code coverage or mutation score and *potential redundant* if they reduce coverage or mutation score. We measure code coverage and mutation score loss using JaCoCo⁵ and PIT⁶, respectively. To compute the losses, we first check out the

⁵<https://github.com/jacoco/jacoco>

⁶<https://pitest.org/>

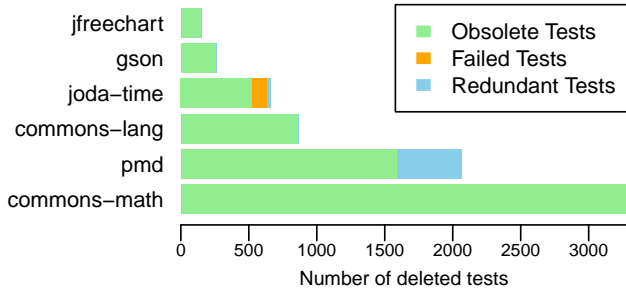


Fig. 4: Reasons for test deletions

TABLE IV: Number of redundant tests (RT) that reduce branch coverage (BC), line coverage (LC), and mutation score (MS)

Project	RT	BC reducing tests	LC reducing tests	MS reducing tests (Total)
commons-lang	4	2	2	2 (3)
commons-math	16	0	0	0 (14)
joda-time	23	0	0	0 (3)
pmd	475	86	101	64 (349)
Total	518	88	103	66 (369)

immediate parent commit of each test deletion commit. We then measure branch coverage, line coverage, and mutation score with and without each redundant test. We calculate the code coverage and mutation score loss (i.e., percentage difference) at the class level. We do not investigate reasons for test deletion in CTS, because it only contains test code.

2) *Findings*: Figure 4 shows the reasons for test deletions in the non-CTS projects. We find that 6,694 (91.4%) of the 7,326 deleted tests are obsolete. All the deleted tests in `gson` and `jfreechart` are obsolete, while 99% of the deleted tests in `commons-lang` and `commons-math` are obsolete. Among 6,694 obsolete tests, 4,164 (62%) are obsolete-parallel and 2,530 (38%) are obsolete-phased. The median interval between production code deletion and the corresponding obsolete-phased test deletion is 1 day, but an obsolete-phased test remains in the codebase for up to 31 days. Recall that we found many tests deleted in some commits in RQ2. We analyze the test deletion commit with the maximum number of deleted tests from each non-CTS project and find that they are all obsolete tests. Developers delete them because the production code is deprecated, rolled back to a previous version, or ported to a more specific project/library.

We find 518 (7%) redundant tests across 111 commits in four projects, 415 (80%) are confirmed redundant and 103 (20%) are potential redundant. Table IV shows the number of deleted redundant tests that reduce line and branch coverage and mutation score. We find that 88 redundant tests reduce both branch and line coverage. Additionally, 15 redundant tests reduce only line coverage. Overall, 103 redundant tests reduce line or branch coverage, and 101 of them are only in `pmd`. Since we measure coverage in the parent commits of the test deletion commits, we manually check whether any new tests are added in the test deletion commits making these

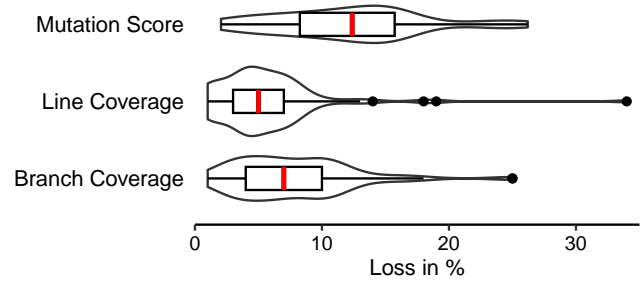


Fig. 5: Coverage and mutation score loss by redundant tests.

```

195 - /**
196 -  * Test the <code>getLocale()</code> method.
197 -  * @see org.joda.time.format.DateTimeFormat#getLocale()
198 -  */
199 - public void testGetLocale() {
200 -     fail("TBD");
201 - }

```

Fig. 6: An assertion failure test from `joda-time` in `0e07ac6`.

103 tests redundant. We do not find any new tests added in the commits. We could not successfully run PIT on the commits containing 149 redundant tests due to configuration issues. Previous studies have also found similar issues with PIT [26], [27]. Table IV shows the results for the remaining 369 redundant tests. We find that 66 redundant tests reduce mutation scores, and 64 of them are only in `pmd`. These 66 tests also reduce both line and branch coverage. Figure 5 shows the line coverage, branch coverage, and mutation score loss due to the removal of the 88, 103, and 66 redundant tests, respectively. The median branch and line coverage loss are 7% and 5%, respectively. However, the removal of a redundant test resulted in up to 25% and 34% branch and line coverage loss, respectively. We also find that the median mutation score loss is 12%, but reaches up to 26%.

We find that 114 (1.6%) deleted tests are failed tests. All of them are in four test deletion commits in `joda-time` and failed due to assertion failures. Upon manual inspection, we find that the tests were added in two commits, one with 107 tests and the other with 7 tests. The tests have no meaningful logic and the failures are intentional. Developers remove 112 of these tests within four days, while one test gets removed after 247 days and another after 276 days. Figure 6 shows an example test from `joda-time`.

3) *Discussion*: Most tests (91.4%) are deleted because their production code is deleted, which is expected as tests become obsolete when the code they test no longer exists. However, the percentage is significantly higher than the 58% found by Pinto et al. in the same projects [1]. Pinto et al. first automatically identified obsolete, hard-to-fix (i.e., failed), and redundant tests and then manually analyzed a few of them. They found that most of the analyzed redundant tests are refactored tests and hard-to-fix tests are obsolete tests. We believe this is why they identified a significantly lower obsolete test percentage than actually exists. Our findings show that a significant percentage (38%) of obsolete tests are not deleted along with

their production code in the same commit. There could be several reasons for this, such as lack of time to run all tests and unawareness of the existence of the tests [28], [29].

The results show that the deleted failed tests are from only one project and all of them are intentional. This suggests the developers may have followed a test-driven development approach [30]. However, if such tests are not promptly followed by feature implementation and updates to the test suite, they become test smells [31], [32]. We do not observe this practice in the other projects.

Although redundant tests are challenging to identify, the results show that developers do identify and delete them. However, 92% of the deleted redundant tests are from only one project. This indicates that it is not a widely adopted practice across projects. However, it is also possible that the other projects do not have redundant tests. The results also show that 20% of the redundant tests reduce code coverage or mutation score. One potential reason for deleting these tests is because they are flaky [33]. We ran each of these tests three times and did not find any flaky tests. It is possible that developers do not focus on code coverage and mutation scores while removing redundant tests. This can be verified only through a user study.

RQ4: 91.4% of the deleted tests are obsolete, among these 62% are obsolete-parallel and 38% are obsolete-phased. 7% of the deleted tests are redundant and 20% of them reduce code coverage or mutation score. Only 1.6% of the deleted tests are failed tests and all of them are intentionally failed.

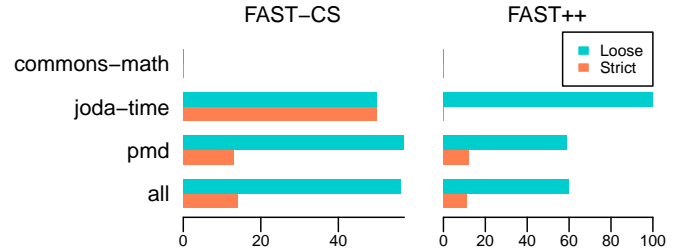
E. RQ5: How many of the developers' deleted redundant tests do TSR approaches identify?

1) *Method:* We select FAST-R [34] to assess the capability of TSR approaches in identifying redundant tests. FAST-R contains four different TSR approaches: FAST++, FAST-CS, FAST-pw, and FAST-all. They are scalable similarity-based approaches. They reduce test suites at the test class level, which covers the majority of the redundant tests identified in RQ4. DELTEST has 426 (82%) redundant tests removed along with 72 test classes in 49 commits from three projects. We pass two inputs to the FAST-R approaches: (1) the original test suite and (2) a test budget (i.e., the reduced test suite size needed in %). We consider the test suite in the direct parent commit of a test deletion commit as the original test suite, resulting in 49 original test suites shown in Table V.

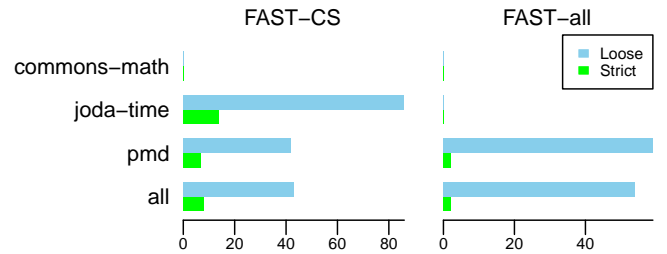
We consider two budget scenarios for each original test suite: *strict* and *loose*. In the strict scenario, we calculate the budget as the percentage of test classes kept in the corresponding test deletion commit. For example, if a developer deletes two test classes in a commit and its parent commit has 10 test classes, the budget is 80%. In the loose scenario, we calculate the budget as the minimum of all strict budgets in a project. Therefore, all original test suites in a project have the same loose budget. For example, as shown in Table V, strict budgets for 46 original test suites in pmd range from 95.45%

TABLE V: Dataset used to evaluate FAST-R approaches

Project	#Original test suites	#Deleted test classes/tests	Budget
commons-math	1	1/9	99.81%
joda-time	2	2/21	99.07%-99.32%
pmd	46	69/396	95.45%-99.88%
	49	72/426	



(a) % of the deleted test classes excluded in the reduced test suites



(b) % of the redundant tests excluded in the reduced test suites

Fig. 7: FAST-R evaluation results

to 99.88%, therefore, the loose budget for pmd is 95.45%. The strict and loose scenarios represent the budget at the commit and project levels. Overall, we generate eight reduced test suites (i.e., four approaches in two budget scenarios) for each original test suite. Since the FAST-R approaches are non-deterministic, we run each approach in each budget scenario on each original test suite 50 times as evaluated originally and select the optimally reduced instance, the instance that excludes the maximum number of redundant test classes.

2) *Findings:* Figure 7 shows the results. Due to the space limitation, we show the results of only two best-performing FAST-R approaches in strict and loose scenarios. The results of the other two approaches in both scenarios are available in our artifacts. Figure 7a shows the percentage of the developers' deleted test classes excluded in the reduced test suites in loose and strict budgets. None of the FAST-R approaches identify the developer's deleted test class in commons-math in both loose and strict budgets. FAST-CS and FAST++ show better performance in both joda-time and pmd. In the strict budgets, FAST-CS performs the best, excluding 50% and 13% of the test classes in joda-time and pmd, respectively. Whereas, in the loose budgets, FAST++ performs the best, excluding 100% and 59% of the test classes in joda-time and pmd, respectively.

Figure 7b shows the percentage of the developers' deleted redundant tests excluded in loose and strict budgets. Since the FAST-R approaches do not exclude the developer's deleted

test class in commons-math, they also do not exclude any redundant tests from the test class. We do not find any one approach performing better in excluding the redundant tests across the projects in strict and loose scenarios. In the strict scenario, FAST-CS and FAST-pw perform the best by excluding 8% and 52% of the redundant tests in pmd and joda-time, respectively. In the loose scenario, FAST++ and FAST-all perform the best by excluding 100% and 59% of the redundant tests in joda-time and pmd, respectively. Overall, across the projects and budget scenarios, FAST-CS performs the best in the loose scenario by identifying 43% of the redundant tests. Although FAST-all does not identify any redundant tests in commons-math and joda-time in the loose scenario, it identifies 54% of the total redundant tests because most redundant tests are present only in pmd.

3) *Discussion*: Although the FAST-R approaches show some potential by identifying up to 54% of the redundant tests, they do not perform consistently at the individual project level. The results also show that the performance of the FAST-R approaches improves significantly when the budget is reduced (i.e., in the loose scenario). We even ran FAST-R with the lowest budget across the projects (95.45%) and found that three of the FAST-R approaches identify the redundant tests in commons-math as well. While a large reduction in the budget may work during regression testing for removing redundant tests temporarily, it may not be suitable for identifying redundant tests for permanent test deletion due to many potential false positives.

RQ5: Not a single FAST-R approach performs consistently across the projects. FAST++ performs better in joda-time by identifying 100% of the redundant tests, while FAST-all performs better in pmd by identifying 59% of the redundant tests.

IV. IMPLICATIONS

A. Identify obsolete tests

The majority (91%) of the developers' deleted tests are obsolete tests. Although these tests can be identified through compilation errors, we found that 38% of them are not deleted along with their production code. This may increase test maintenance costs [35]. Automated just-in-time techniques that identify obsolete tests and notify developers to delete them along with the production code can help developers maintain clean test suites and facilitate production and test code co-evolution. The fact that most (83%) test deletions occur at the test class level may help improve the efficiency of these techniques.

B. Identify redundant tests

FAST-R approaches have limited accuracy in identifying developers' deleted redundant tests across projects. However, they are scalable and have a high recall under a reasonable budget (95.45%). Therefore, the approaches can be used as a filtering step to identify candidate redundant tests. We found that most (80%) of the developers' deleted redundant tests

do not reduce code coverage or mutation scores. Therefore, to identify redundant tests for permanent test deletion, TSR approaches or new techniques can further analyze the candidate redundant tests for code coverage and mutation score loss. Note that calculating and maintaining code coverage and mutation score is expensive [10], [36], [37]. Therefore, performing these tasks on a limited number of candidate redundant tests can improve the performance of the techniques.

C. Build, evaluate, and compare test deletion techniques

We contributed DELTEST, the first manually confirmed dataset of developers' deleted obsolete and redundant tests. Machine learning techniques for predicting obsolete and redundant tests can use this dataset for training [35], [13]. Test deletion techniques can also use the dataset as ground truth to evaluate their effectiveness in identifying obsolete and redundant tests. Moreover, the dataset can facilitate comparisons of the effectiveness of test deletion techniques.

D. Mine developers' deleted tests

Identifying refactored tests is crucial in mining developers' deleted tests. If we do not filter out all the refactored tests, the results will have too many false positives, as found in a previous study [1] and validated by our results. We used RefactoringMiner [23], a state-of-the-art tool for detecting test refactorings. We manually reviewed 200 of the 5,230 test refactorings detected in Section II-D and found that RefactoringMiner correctly identified all of them as test refactorings. However, we found during our manual analysis in Section II-E that RefactoringMiner failed to identify 3,210 test refactorings, which is a significant number. Other studies have also found the limitation of RefactoringMiner in identifying test refactorings [38]. Researchers can use our dataset to identify and address the limitations of test refactoring detection techniques, which can help test deletion and refactoring studies.

V. THREATS TO VALIDITY

A. Construct validity:

Developers can partially delete a test by deleting its statements [39], [40], [41]. In this study, we did not consider test deletion at the statement level. However, most of the existing test evolution and TSR techniques can use our dataset and empirical results, because they analyze tests at the individual test or class level.

B. Internal validity

We performed several automated steps to create DELTEST. In the process, we may have missed some deleted tests. To mitigate the threat, we manually validated some of the results after each automated step, ensuring tools and techniques used in the step do not have any flaws. For example, we manually validated 200 of the 5,230 refactored tests identified by RefactoringMiner and found the results were correct. We manually reviewed code changes and commit messages to confirm the deleted tests. However, we found that sometimes test deletion is not obvious from the code changes or commit

messages. To mitigate this threat, two authors independently reviewed each candidate deleted test.

C. External validity

DELTEST, on which the empirical results are based, has deleted tests from only seven open-source Java projects. However, six of the projects are commonly used in test suite evolution [3], [18], [19], [20], [16] and TSR [34] studies. The remaining one is an open-source large-scale test suite developed and maintained internally by Google, which provides diversity to DELTEST in terms of types of projects. We selected FAST-R approaches [34] as representative TSR approaches to assess the capability of identifying developers' deleted redundant tests, mainly because they are scalable approaches that do not require any information (e.g., code coverage and mutation) about tests.

VI. RELATED WORK

Two categories of work are particularly related to our study, test suite evolution and test suite reduction, which we discuss in this section.

A. Test suite evolution

Several studies have investigated the co-evolution of production and test code. Zaidman et al. analyzed added and changed production and test code to understand their co-evolution nature (i.e., synchronous or phased), developers' testing strategies, testing effort, and test quality (i.e., coverage) [42], [2]. Marinescu et al. examined how programs evolve in terms of code, test, and coverage, including the co-evolution of production and test code [43]. Although these studies analyze code changes, they do not delve into test deletion. Lubson et al. investigated whether association rule mining can be used to find evidence and extent of production and test code co-evolution [44]. Marsavina et al. later used association rule mining to examine how test code evolves with different types of changes in production code [3]. They identified six co-evolution patterns, including the deletion of test methods and classes when corresponding production methods and classes are deleted or modified. Shimmit et al. performed a literature search and analyzed open-source repositories to identify co-evolution patterns [16]. They also proposed test addition and modification remedies for the identified production code change patterns. Levin et al. investigated how production code maintenance activities (i.e., corrective, perfective, and adaptive) and semantic code changes (e.g., removal of a class or method) co-relate with test maintenance activities and test counts [17]. They found that the removal of a production class, method, or statement increases the odds of test deletion. Co-evolution studies make various assumptions while identifying co-evolution commits. Sun et al. found that these assumptions introduce noises in co-evolution datasets, resulting in inaccurate results [45]. They also proposed an approach to identify co-evolution artifacts.

While some of these co-evolution studies provide evidence of test deletion and its reason (i.e., production code removal),

they focus only on obsolete tests and do not report how many and how often developers delete tests. Only Pinto et al. reported this in their test suite evolution study and specifically investigated the reasons for test deletion, although their main focus was test modification/repair [1]. In contrast to Pinto et al.'s test deletion study, our study is more comprehensive in terms of (1) subjects: our study additionally includes a large-scale privately developed and managed test suite, (2) data points: we investigate 449,592 commits in comparison to 88 versions investigated by Pinto et al., and (3) datasets: our results are based on 24,431 manually confirmed deleted tests in comparison to 2,541 unconfirmed deleted tests identified and investigated by Pinto et al. We additionally investigate test deletion granularity and test suite size reduction.

B. Test suite reduction

Several TSR approaches have been proposed in the last three decades [46], [47], [48]. They mainly use greedy [9], [49], clustering [34], search-based [50], or hybrid [51] algorithms that rely on test code similarity [34], code coverage [9], [52], or fault coverage [52], [53] to identify redundant tests and remove them from test suites. In this work, we do not propose a new TSR approach. Instead, we evaluate existing TSR approaches to assess their capabilities in identifying redundant tests for permanent test deletion.

Khan et al. performed a literature review to understand the quality of experiments in TSR studies [48]. They made several recommendations to improve TSR experiments. We also aim to improve TSR experiments through this work. However, we focus on permanent test deletion and provide a dataset and empirical evidence to improve TSR experiments. Previous TSR studies mainly use test suite size reduction (higher is better) and fault detection capability loss (lower is better) properties of the reduced test suites to assess the effectiveness of TSR approaches [54], [55], [56]. In contrast, we evaluate the effectiveness of FAST-R approaches by checking how many of the developers deleted redundant tests these approaches identify and remove them from test suites. This evaluation criteria is critical to using TSR approaches for permanent test deletion. TSR approaches have never been evaluated on developers' deleted redundant tests.

Several empirical studies have been performed to compare different TSR approaches [57], [58], [59], [60]. We also evaluate four different clustering-based TSR approaches from FAST-R. However, our main goal in this work is to assess the capabilities of TSR approaches in identifying redundant tests for permanent test deletion.

VII. CONCLUSION

We conducted an empirical study to understand how and why developers delete tests. To do this, we analyzed 449,592 commits from seven open-source Java projects and created DELTEST, a dataset containing 24,431 manually confirmed deleted tests. We found that test deletion is an infrequent software development activity. Both the frequency and number of deleted tests vary significantly across projects, suggesting

that test deletion is more likely driven by project-specific needs, such as the pace of code evolution, rather than a broader development cycle. Most tests were deleted because they were obsolete, although we also found failed and redundant tests removed in certain projects. However, these were mostly limited to a single project, and FAST-R approaches have the potential to identify redundant tests.

REFERENCES

- [1] 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*, 2012, pp. 1–11.
- [2] A. Zaidman, B. Van Rompaey, S. Demeyer, and A. Van Deursen, "Mining software repositories to study co-evolution of production & test code," in *2008 1st international conference on software testing, verification, and validation*. IEEE, 2008, pp. 220–229.
- [3] C. Marsavina, D. Romano, and A. Zaidman, "Studying fine-grained co-evolution patterns of production and test code," in *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2014, pp. 195–204.
- [4] D. J. Kim, N. Tsantalis, T.-H. P. Chen, and J. Yang, "Studying test annotation maintenance in the wild," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 62–73.
- [5] T. Xie, D. Notkin, and D. Marinov, "Rostra: A framework for detecting redundant object-oriented unit tests," in *Proceedings. 19th International Conference on Automated Software Engineering, 2004*. IEEE, 2004, pp. 196–205.
- [6] H. K. Leung and L. White, "Insights into regression testing (software testing)," in *Proceedings. Conference on Software Maintenance-1989*. IEEE, 1989, pp. 60–69.
- [7] A. Labuschagne, L. Inozemtseva, and R. Holmes, "Measuring the cost of regression testing in practice: A study of java projects using continuous integration," in *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, 2017, pp. 821–830.
- [8] G. Rothermel, S. Elbaum, A. G. Malishevsky, P. Kallakuri, and X. Qiu, "On test suite composition and cost-effective regression testing," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 13, no. 3, pp. 277–331, 2004.
- [9] M. J. Harrold, R. Gupta, and M. L. Soffa, "A methodology for controlling the size of a test suite," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 2, no. 3, pp. 270–285, 1993.
- [10] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, "Taming google-scale continuous testing," in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 2017, pp. 233–242.
- [11] A. A. Philip, R. Bhagwan, R. Kumar, C. S. Maddila, and N. Nagppan, "Fastlane: Test minimization for rapidly deployed large-scale online services," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 408–418.
- [12] R. Greca, B. Miranda, and A. Bertolino, "State of practical applicability of regression testing research: A live systematic literature review," *ACM Computing Surveys*, vol. 55, no. 13s, pp. 1–36, 2023.
- [13] D. Hao, T. Lan, H. Zhang, C. Guo, and L. Zhang, "Is this a bug or an obsolete test?" in *European Conference on Object-Oriented Programming*. Springer, 2013, pp. 602–628.
- [14] A. Vahabzadeh, A. M. Fard, and A. Mesbah, "An empirical study of bugs in test code," in *2015 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 2015, pp. 101–110.
- [15] M. Skoglund and P. Runeson, "A case study on regression test suite maintenance in system evolution," in *20th IEEE International Conference on Software Maintenance, 2004. Proceedings*. IEEE, 2004, pp. 438–442.
- [16] S. Shimmi and M. Rahimi, "Patterns of code-to-test co-evolution for automated test suite maintenance," in *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2022, pp. 116–127.
- [17] 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, pp. 35–46.
- [18] M. Mirzaaghaei, F. Pastore, and M. Pezzè, "Automatic test case evolution," *Software Testing, Verification and Reliability*, vol. 24, no. 5, pp. 386–411, 2014.
- [19] A. R. Chen, T.-H. P. Chen, and S. Wang, "T-evos: A large-scale longitudinal study on ci test execution and failure," *IEEE Transactions on Software Engineering*, 2022.
- [20] L. Vidács and M. Pinzger, "Co-evolution analysis of production and test code by learning association rules of changes," in *2018 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*. IEEE, 2018, pp. 31–36.
- [21] D. Spadini, M. Aniche, and A. Bacchelli, "Pydriller: Python framework for mining software repositories," in *Proceedings of the 2018 26th ACM Joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2018, pp. 908–911.
- [22] B. Van Rompaey and S. Demeyer, "Establishing traceability links between unit test cases and units under test," in *2009 13th European Conference on Software Maintenance and Reengineering*. IEEE, 2009, pp. 209–218.
- [23] N. Tsantalis, A. Ketkar, and D. Dig, "Refactoringminer 2.0," *IEEE Transactions on Software Engineering*, vol. 48, no. 3, pp. 930–950, 2020.
- [24] J. Cohen, "A coefficient of agreement for nominal scales," in *Educational and Psychological Measurement*, vol. 20, 1960, pp. 37–46. [Online]. Available: <https://api.semanticscholar.org/CorpusID:15926286>
- [25] A. Najafi, W. Shang, and P. C. Rigby, "Improving test effectiveness using test executions history: An industrial experience report," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2019, pp. 213–222.
- [26] M. Ojdanic, E. Soremekun, R. Degiovanni, M. Papadakis, and Y. Le Traon, "Mutation testing in evolving systems: Studying the relevance of mutants to code evolution," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 1, 2023.
- [27] M. Kintis, M. Papadakis, A. Papadopoulos, E. Valvis, N. Malevris, and Y. L. Traon, "How effective are mutation testing tools? an empirical analysis of java mutation testing tools with manual analysis and real faults," *Empirical Software Engineering*, vol. 23, pp. 2426 – 2463, 2017.
- [28] V. Hurdugaci and A. Zaidman, "Aiding software developers to maintain developer tests," in *2012 16th European Conference on Software Maintenance and Reengineering*. IEEE, 2012, pp. 11–20.
- [29] M. Beller, G. Gousios, A. Panichella, and A. Zaidman, "When, how, and why developers (do not) test in their ides," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 179–190.
- [30] S. Romano, F. Zampetti, M. T. Baldassarre, M. Di Penta, and G. Scanniello, "Do static analysis tools affect software quality when using test-driven development?" in *Proceedings of the 16th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*. New York, NY, USA: Association for Computing Machinery, 2022, p. 80–91. [Online]. Available: <https://doi.org/10.1145/3544902.3546233>
- [31] S. Reichhart, T. Gırba, and S. Ducasse, "Rule-based assessment of test quality," *Journal of Object Technology*, vol. 6, pp. 231–251, 2007. [Online]. Available: <https://api.semanticscholar.org/CorpusID:3852703>
- [32] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De 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*, ser. ASE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 4–15. [Online]. Available: <https://doi.org/10.1145/2970276.2970340>
- [33] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, "Understanding flaky tests: the developer's perspective," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 830–840. [Online]. Available: <https://doi.org/10.1145/3338906.3338945>
- [34] E. Cruciani, B. Miranda, R. Verdecchia, and A. Bertolino, "Scalable approaches for test suite reduction," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 419–429.
- [35] S. Wang, M. Wen, Y. Liu, Y. Wang, and R. Wu, "Understanding and facilitating the co-evolution of production and test code," in *2021*

- IEEE International conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2021, pp. 272–283.
- [36] M. Machalica, A. Samykin, M. Porth, and S. Chandra, “Predictive test selection,” in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2019, pp. 91–100.
- [37] S. Elbaum, G. Rothermel, and J. Penix, “Techniques for improving regression testing in continuous integration development environments,” in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 235–245.
- [38] L. Martins, H. Costa, M. Ribeiro, F. Palomba, and I. Machado, “Automating test-specific refactoring mining: A mixed-method investigation,” in *2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2023, pp. 13–24.
- [39] A. M. Arash Vahabzadeh, Andrea Stocco, “Fine-grained test minimization,” in *In Proceedings of the International Conference on Software Engineering*. IEEE, 2018, pp. 1–10.
- [40] M. A. Alipour, A. Shi, R. Gopinath, D. Marinov, and A. Groce, “Evaluating non-adequate test-case reduction,” in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 16–26.
- [41] S. Herfert, J. Patra, and M. Pradel, “Automatically reducing tree-structured test inputs,” in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 861–871.
- [42] A. Zaidman, B. Van 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,” *Empirical Software Engineering*, vol. 16, no. 3, pp. 325–364, 2011. [Online]. Available: <https://doi.org/10.1007/s10664-010-9143-7>
- [43] P. Marinescu, P. Hosek, and C. Cadar, “Covrig: A framework for the analysis of code, test, and coverage evolution in real software,” in *Proceedings of the 2014 international symposium on software testing and analysis*, 2014, pp. 93–104.
- [44] Z. Lubsen, A. Zaidman, and M. Pinzger, “Using association rules to study the co-evolution of production and test code,” in *2009 6th IEEE International Working Conference on Mining Software Repositories*, 2009, pp. 151–154.
- [45] W. Sun, M. Yan, Z. Liu, X. Xia, Y. Lei, and D. Lo, “Revisiting the identification of the co-evolution of production and test code,” *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 6, pp. 1–37, 2023.
- [46] H. Do, “Recent advances in regression testing techniques,” *Advances in computers*, vol. 103, pp. 53–77, 2016.
- [47] S. Yoo and M. Harman, “Regression testing minimization, selection and prioritization: a survey,” *Software testing, verification and reliability*, vol. 22, no. 2, pp. 67–120, 2012.
- [48] S. U. R. Khan, S. P. Lee, N. Javaid, and W. Abdul, “A systematic review on test suite reduction: Approaches, experiment’s quality evaluation, and guidelines,” *IEEE Access*, vol. 6, pp. 11 816–11 841, 2018.
- [49] T. Chen and M. Lau, “A new heuristic for test suite reduction,” *Information and Software Technology*, vol. 40, no. 5, pp. 347–354, 1998. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0950584998000500>
- [50] J. Geng, Z. Li, R. Zhao, and J. Guo, “Search based test suite minimization for fault detection and localization: A co-driven method,” in *International Symposium on Search Based Software Engineering*, 2016. [Online]. Available: <https://api.semanticscholar.org/CorpusID:41295271>
- [51] S. Yoo and M. Harman, “Using hybrid algorithm for pareto efficient multi-objective test suite minimisation,” *Journal of Systems and Software*, vol. 83, no. 4, pp. 689–701, 2010.
- [52] H.-Y. Hsu and A. Orso, “Mints: A general framework and tool for supporting test-suite minimization,” in *2009 IEEE 31st international conference on software engineering*. IEEE, 2009, pp. 419–429.
- [53] M. Polo Usaola, P. Reales Mateo, and B. Pérez Lamancha, “Reduction of test suites using mutation,” in *Fundamental Approaches to Software Engineering: 15th International Conference, FASE 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24-April 1, 2012. Proceedings 15*. Springer, 2012, pp. 425–438.
- [54] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong, “An empirical study of the effects of minimization on the fault detection capabilities of test suites,” in *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. IEEE, 1998, pp. 34–43.
- [55] W. Wong, J. Horgan, A. Mathur, and A. Pasquini, “Test set size minimization and fault detection effectiveness: a case study in a space application,” in *Proceedings Twenty-First Annual International Computer Software and Applications Conference (COMPSAC’97)*, 1997, pp. 522–528.
- [56] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur, “Effect of test set minimization on fault detection effectiveness,” *Software: Practice and Experience*, vol. 28, no. 4, pp. 347–369, 1998.
- [57] C. Coviello, S. Romano, and G. Scanniello, “An empirical study of inadequate and adequate test suite reduction approaches,” in *Proceedings of the 12th ACM/IEEE international symposium on empirical software engineering and measurement*, 2018, pp. 1–10.
- [58] C. Coviello, S. Romano, G. Scanniello, A. Marchetto, G. Antoniol, and A. Corazza, “Clustering support for inadequate test suite reduction,” in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 95–105.
- [59] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid, “An empirical study of junit test-suite reduction,” in *2011 IEEE 22nd International Symposium on Software Reliability Engineering*, 2011, pp. 170–179.
- [60] A. Shi, A. Gyori, S. Mahmood, P. Zhao, and D. Marinov, “Evaluating test-suite reduction in real software evolution,” in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 84–94.