Reducing Test Runtime by Transforming Test Fixtures

Chengpeng Li
Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX, USA
chengpengli@utexas.edu

Abdelrahman Baz
Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX, USA
ambaz@utexas.edu

August Shi
Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX, USA
august@utexas.edu

ABSTRACT

Software testing is a fundamental part of software development, but the cost of running tests can be high. Existing approaches to speed up testing such as test-suite reduction or regression test selection aim to run only a subset of tests from the full test suite, but these approaches run the risk of missing to run some key tests that are needed to detect faults in the code.

We propose a new technique to transform test code to speed up test runtime while still running all the tests. The insight is that testing frameworks such as JUnit for Java projects allow for developers to define test fixtures, i.e., methods that run before or after every test to setup or teardown test state, but these test fixtures need not be called all the time before/after each test. It may be sufficient to do the setup and teardown once at the beginning and end, respectively, of all tests. Our technique, TestBoost, transforms the test fixtures within a test class to instead run once before/after all tests in the test class, thereby running the test fixtures less frequently while still running all tests and ensuring that tests all still pass, as they did before. Our evaluation on 697 test classes from 34 projects shows that on average we can reduce the runtime per test class by 28.39% for the cases with positive significant improvement. Using these transformed test classes can result in an average 18.24% reduction per test suite runtime. We find that the coverage of the transformed test classes changes by <1%, and when we submitted 15 pull requests, 9 have already been merged.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Regression testing, test fixtures, testing speedup

ACM Reference Format:
Chengpeng Li, Abdelrahman Baz, and August Shi. 2024. Reducing Test Runtime by Transforming Test Fixtures. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3691620.3695541

1 INTRODUCTION

Regression testing is the practice of rerunning an existing regression test suite after every change to check whether those changes break existing functionality [55]. While widely practiced, regression testing is costly in terms of test runtime, as test suites contain many tests, and those tests have to be run after every change, which happens frequently [1, 12, 16, 38, 39]. For example, Google reported that their regression testing system on average handles 800K builds and 150 millions test runs per day [39].

Prior work proposed techniques to speed up regression testing, such as test-suite reduction or regression test selection [55]. Both these approaches aim to reduce the cost of testing by running only a subset of the full test suite. However, by running only a subset of tests, these approaches run the risk of missing to detect faults if they do not run the key test(s) that could detect those faults. For example, test-suite reduction creates a reduced test suite by removing tests considered redundant w.r.t. metrics like code coverage [9, 10, 18, 24, 37, 44, 48, 58], but subsequent work found these reduced test suites miss to detect real faults in future versions of the code [49]. Regression test selection selects to run tests affected by the code changes, meaning it should ideally not miss to select tests that can detect newly-introduced faults [16, 29, 34, 57], but these techniques may still have limitations and miss to select the key fault-detecting tests [60].

Ideally, we would run all tests without skipping any, but run them faster. Our insight is that tests often contain test fixtures, which are setup and teardown methods that the testing framework runs before/after each test as a means to set up the state for the test to run or to clean up changes to state shared between tests. For example, in the JUnit 4 testing framework, these test fixtures are the methods annotated with `@Before`, indicating they run before each test, or annotated with `@After`, indicating they run after each test. However, tests may not always need these test fixtures to run so frequently. Some tests may not actually rely on state modified in the test fixtures, and some of that setup/teardown logic may be fine to run just once at the beginning or at the end of all tests run. In other words, we can save time by transforming these test fixtures to run less often, saving overall test runtime. The reduction in runtime can be high if the test fixtures are performing expensive operations, e.g., setting up a database, and there are many tests to run (meaning these expensive test fixtures run often).

We propose TestBoost, a technique for reducing test runtime by transforming test fixtures. Focusing on Java and the JUnit testing framework, we target test fixtures that run before/after individual tests defined in a test class to instead run once at the beginning and/or at the end of the test class, before/after all tests in the test class run. We still want to run the test fixtures at least once w.r.t. all tests within a test class, ensuring proper isolation in shared state

ASE '24, October 27-November 1, 2024, Sacramento, CA, USA

ACM ISBN 979-8-4007-1248-7/24/10
https://doi.org/10.1145/3691620.3695541

between test classes. Our transformation for a test class involves configuring the test fixture methods to run once per test class, changing the scope of fields and methods to accommodate the test fixture changes, as well transforming the other test classes related to the target test class based on class hierarchy.

We make further transformations on test classes to ensure their tests do not fail due to becoming order-dependent flaky tests (OD tests). An OD test is a test whose outcome changes due to the order in which the tests are run [25, 35, 59]. The reason why these tests fail in different orders is because they are dependent on other tests with which they share state. Test fixtures are commonly used to set up or clean that shared state between tests, so transforming them to not run between tests can cause previously non-OD tests to become OD tests. To handle these cases, TestBoost also reruns tests in the transformed test classes in different orders to see whether any of them pass and fail in different orders. If TestBoost finds OD tests, it makes additional transformations to the test class by injecting into those tests additional calls to the test fixtures. Essentially, we aim to emulate the same behavior as when test fixtures run in-between each test, but targeting only the tests that need it.

We evaluate TestBoost by transforming 697 test classes, containing a total of 38430 test methods, spread across 34 open-source Java projects from GitHub. Our evaluation shows that TestBoost can successfully transform 500 test classes while ensuring tests still pass after transformation. When we measure the runtime reduction using the 500 transformed test classes, we find that 169 of these test classes provide a statistically significant positive reduction in runtime, with an average reduction of 28.39% per test class. By replacing the tests in the test suite with these 500 transformed test classes, the test suite runtime on average reduces by 18.24%. Furthermore, we check the difference in code coverage after transformation, finding the coverage on averages changes <1% per test class. We additionally submit 15 pull requests to developers with our changes, of which 9 have been merged. An artifact containing TestBoost code and the results of this paper is available online [4].

This paper makes the following main contributions:

- We identify transforming test fixtures as a means to reduce test runtime while still running all tests.
- We present TestBoost to transform test fixtures by having them run only once before/after all tests in a test class.
- We evaluate TestBoost on a dataset of 697 test classes from 34 open-source projects. TestBoost successfully transforms 500 test classes. When the transformation provides a positive significant reduction in runtime, that reduction in runtime is on average 28.39% per test class, leading to an average reduction of 18.24% in the test suite.
- We find the transformed test classes to have on average <1% change in coverage from before transformation. We also submitted 15 pull requests, with 9 merged.

2 EXAMPLE

Figure 1 shows an example of a JUnit test class from project `elasticjob/elastic-job-lite`, including the changes before and after transforming the test fixtures in the test class. This test class contains three tests along with three test fixtures, which are the methods annotated with `@BeforeClass` (Line 8), `@Before` (Line 13), and `@After` (Line 19). These test fixtures set up and clean up any state that tests need to run. In this example, the static method annotated with `@BeforeClass`, `init()`, runs once before all tests in the test class. The method annotated with `@Before`, `setUp()`, runs before every test in the test class, and the method annotated with `@After`, `teardown()`, runs after every test in the test class. More specifically, `init()` runs once to start the `EmbedTestingServer` for all the tests, `setUp()` creates and initializes a new `ZooKeeperRegistryCenter` instance before each test, while `teardown()` closes this instance after each test finishes.

```
1   public final class OneOffJobBootstrapTest {
2     private static final ZookeeperConfiguration ZKC =
3       new ZookeeperConfiguration(...);
4     private static final int SHARDING_TOTAL_COUNT = 3;
5 +  private static ZookeeperRegistryCenter zkRegCenter;
6 -  private ZookeeperRegistryCenter zkRegCenter;
7
8     @BeforeClass public static void init() {
9       EmbedTestingServer.start();
10 +    zkRegCenter = new ZookeeperRegistryCenter(ZKC);
11 +    zkRegCenter.init();
12    }
13 -  @Before public void setUp() {
14 -    zkRegCenter = new ZookeeperRegistryCenter(ZKC);
15 -    zkRegCenter.init();
16 -  }
17
18 +  @AfterClass public static void teardown() {
19 -  @After public void teardown() {
20      zkRegCenter.close();
21    }
22
23    // Tests
24    ...
25  }
```

Figure 1: Simplified example test class from `elasticjob/elastic-job-lite` with transformations.

Our intuition is that the test fixtures that run before and after each test, namely `setUp()` and `teardown()`, may not need to run so often. The tests may still pass even if they do not use a fresh new `ZooKeeperRegistryCenter` instance and close that instance every time the test starts and ends, respectively. These test fixtures may only need to run once before and once after all tests in the test class, running less often and thereby saving overall test runtime.

To transform the test fixtures to run less frequently, we transform the `@Before` method into a `@BeforeClass` method so it runs once at the beginning of the test class before all tests, and we transform the `@After` method into a `@AfterClass` method so it runs once at the end of the test class after all tests. In this example, because a `@BeforeClass` method already exists, we instead merge the code from the original `@Before` method with the existing `@BeforeClass` method to ensure that code runs after what was the original `@BeforeClass` method code.

In addition, test fixtures that are annotated with `@BeforeClass` and `@AfterClass` need to be static methods, so we change the signature of `teardown()` to be static. Furthermore, instance fields like `zkRegCenter` need to be converted to static as well (Line 5).

When we transform the test fixtures and update the fields to become static, the tests in the test class still pass when run. Further, the transformed test class runs faster than the original test class, obtaining a runtime improvement of 48.39%.

```
# testclass: the test class to transform
# Returns the modified test classes
def TestBoost(testclass):
  # Transform test fixtures from test classes
  mod_testclases, compile_status = Transformer(testclass)
  # Return original test class if cannot transform
  if not compile_status == SUCCESS:
    return [testclass]

  # Check whether transformed test class has failing tests
  failedtests = ODChecker(mod_testclasses)
  if len(failedtests) == 0:  # Tests pass in all orders
    return mod_testclasses

  # Inject setup/teardown calls into failing tests
  mod_testclasses, run_status = Injector(mod_testclasses, failedtests)
  if run_status == SUCCESS:
    return mod_testclasses
  else:
    return [testclass]
```

Figure 2: High-level pseudocode.

3 TESTBOOST

TestBoost aims to automatically transform the test fixtures within a test class to ensure they run less frequently, reducing test runtime. The use scenario for TestBoost is that a developer can use it to transform the test classes in their test suite to run faster in the future. TestBoost takes as input a test class to transform and outputs all test classes that TestBoost transforms. The reason TestBoost would transform more than just the input test class is when that test class is a subclass or superclass of other test classes, so those other test classes may need to be transformed as well.

TestBoost is divided into three main components: Transformer, OD-Checker, and Injector. Figure 2 shows pseudocode illustrating how TestBoost calls the different components to transform a test class. TestBoost first uses Transformer to transform test fixtures of the test class to be static, including the related test classes due to class hierarchy. If the transformed test classes still compile, TestBoost then uses OD-Checker to see whether all tests pass when run in any order within each test class, successfully returning the transformed test classes if so. Otherwise, TestBoost relies on Injector to transform a test class further by modifying failing tests to explicitly call the test fixtures to reset state. While test fixtures get called more often in this scenario, ideally not all tests need to call test fixtures for them to all pass. If Injector can make the tests pass in any order, we consider TestBoost to be successful.

3.1 Transformer

Given a test class, the Transformer transforms the test fixtures in the test class such that they execute only once instead of every time before/after each test in the test class. We rely on JavaParser [2], an existing tool for parsing and transforming Java files, to perform this transformation on the test file that contains the test class. More specifically, in the case of JUnit 4 tests, we change the `@Before` annotated methods to be annotated with `@BeforeClass`, so the methods run just once before all tests in the test class, and we change the `@After` annotated methods to be annotated with `@AfterClass`, so the methods run just once after all tests in the test class. For JUnit 5, we perform a similar transformation, except we change `@BeforeEach` to `@BeforeAll` and `@AfterEach` to `@AfterAll`.

```
1  # testclass: the test class to transform
2  # Returns the transformed test classes and status
3  def Transformer(testclass):
4    mod_testclasses = set()
5
6    # Parse hierarchy of current test class
7    tc_tree = parse_hierarchy_tree(testclass)
8    super_testclasses = get_ancestors(testclass, tc_tree)
9    sub_testclasses = get_descendants(testclass, tc_tree)
10
11   # Plan to process all classes related to testclass
12   classes_to_process = super_testclasses + [testclass] + sub_testclasses
13   topological_sort(classes_to_process, tc_tree)
14
15   while len(classes_to_process) > 0:
16     curr = classes_to_process.pop()
17
18     # Find the methods/fields to update annotations,
19     # e.g., @Before becomes @BeforeClass
20     methods, fields = update_anno(curr)
21     if len(methods) > 0 or len(fields) > 0:
22       mod_testclasses.add(curr)
23
24       # Convert methods and fields to static
25       while len(methods) > 0 or len(fields) > 0:
26         new_methods = set()
27         new_fields = set()
28         for m in methods:
29           make_static(m)
30           mod_testclasses.add(get_declaringclass(m))
31           new_methods |= get_referenced_methods(m)
32           new_methods |= get_override_methods(m, tc_tree)
33           new_fields |= get_referenced_fields(m)
34         for f in fields:
35           make_static(f)
36           mod_testclasses.add(get_declaringclass(f))
37           new_methods |= get_referenced_methods(f)
38           new_fields |= get_referenced_fields(f)
39         fields = new_fields
40         methods = new_methods
41
42       new_children = get_descendants(curr, tc_tree)
43       classes_to_process |= new_children
44
45     # Re-sort classes in case new ones got added
46     topological_sort(classes_to_process, tc_tree)
47
48   # Re-compile all the modified test classes
49   status = compile(mod_testclasses)
50   return mod_testclasses, status
```

Figure 3: Transformer pseudocode.

Figure 3 shows pseudocode representing the Transformer workflow. First, Transformer creates a class hierarchy tree involving the test class to be transformed, which can include all the test classes it inherits from as well as those that inherit from it (Lines 7-9). Transformer iterates through all these classes to check which ones need to be transformed. It goes through these classes in a topological ordering, to ensure the classes on the top of the class hierarchy get processed first (Line 13). For each test class to be processed, Transformer checks whether that test class contains any test fixtures that should be transformed; the `update_anno` function automatically rewrites the annotations appropriately based on the testing framework, e.g., methods annotated with `@Before` in JUnit 4 get rewritten to `@BeforeClass` (Line 20). We also change the annotations of fields annotated with `@Rule`, which are fields that JUnit sets up before each test and resets after each test finishes. We change such fields to be static and annotated with `@ClassRule`, so they are set up and reset before/after all tests in a test class.

If the test class has test fixtures to be transformed, it iterates through all the test fixtures and fields to determine whether they reference some other methods and fields, e.g., a test fixture calls other helper methods. Since the test fixtures will become static, they can no longer reference non-static instance methods or fields, so referenced methods and fields need to also become static. We use the functions `get_referenced_methods` and `get_referenced_fields` to get the referenced methods and fields, respectively. Transformer iteratively uses these functions to find all referenced methods and fields referenced, making all of them static (Lines 25-40). Note that these methods/fields may be declared in some superclass. We directly change these methods/fields in the class in which they are declared. We add these declaring classes into the set of modified test classes that `Transformer` eventually returns. We also remove the `final` keyword from fields, so they can be modified in the test fixture when made static. Normally, as instance fields, their values are reset between tests due to JUnit creating new test class instances for each test, and we want to allow that to still be possible in the case of needing to inject calls to test fixtures (Section 3.3). If the field is a mock object for Mockito, meaning it is annotated with `@Mock`, Transformer needs to move the assignment to the `@BeforeClass` or `@BeforeAll` method to ensure it is assigned properly as a static field. Furthermore, we find all methods in subclasses that override any transformed methods and schedule them to be transformed as well. Finally, if the test class at this iteration is transformed in any way, we get all other test classes that inherit from it so we can transform those as well (Lines 42-43). We include these subclasses because their functionality may depend on this transformed superclass, so we need to transform them, e.g., making their methods static. We re-sort the classes in topological order to always ensure processing the classes high on the hierarchy tree first.

If the test class contains any existing `@BeforeClass`/`@BeforeAll` or `@AfterClass`/`@AfterAll` methods, we merge the transformed test fixtures into those existing methods. For transformed `@Before`/`@BeforeEach` methods, we take all their statements and append them to the end of the existing `@BeforeClass`/`@BeforeAll` method (we assume at most only one such method in the test class). Merging the statements this way ensures that all statements from the `@Before`/`@BeforeEach` method execute after the statements from the original `@BeforeClass`/`@BeforeAll` method, matching semantics of the original test class. We perform a similar transformation for `@After`/`@AfterEach` methods, except we prepend the statements into the existing `@AfterClass`/`@AfterAll` method to match the original teardown semantics.

We also handle some additional minor transformations. If we encounter any usage of `this` as a caller object, e.g., `this.call()`, we change the method call to instead use the test class name, e.g., `TestClass.call()`. If we see `this` being used in the form of `this.getClass()`, we replace that specifically with the name of the test class with suffix `.class` to obtain a `Class` instance referring to the test class. If `this` is used not as a caller object, e.g., `MockitoAnnotations.initMocks(this)`, we change the `this` to be a new instance of the test class (test classes must have a no-argument constructor, which we can use to construct instances). Finally, if the call uses `super` to reference a superclass, we replace it with the name of the superclass to access its static method.

```
1  # test classes: transformed test classes to run
2  def ODChecker(testclasses):
3    od_tests = set()
4    for t in testclasses:
5      test_orders = systematically_get_orders(t)
6
7      for order in test_orders:
8        failed_tests = extract_failed_tests(order)
9        od_tests |= failed_tests
10
11   return od_tests
```

Figure 4: OD-Checker pseudocode.

Once we have all the modified test classes, we compile all of them to ensure our changes are valid. `Transformer` returns the set of modified test classes along with that compilation status. If compilation fails, the high-level algorithm would stop and not transform the test class, returning the original test class (Line 7 in Figure 2).

3.2 OD-Checker

Due to the nature of the transformations, tests within a transformed test class may pass/fail when run in different orders, i.e., they now have order-dependent flaky test (OD tests) due to shared state dependencies between tests [25, 35, 50, 59]. Note that the tests may only be dependent on other tests in the same test class, and not across test classes, because our transformations still make sure test fixtures get executed before/after each test class, ensuring isolation between test classes in terms of shared state.

Figure 4 shows pseudocode illustrating OD-Checker, which detects OD tests among the transformed test classes. OD-Checker relies on prior work by Li et al. to systematically search for orders that can detect OD tests [31] (Line 5). This approach generates the minimal set of orders where for every pair of tests, there is an order where one test in that pair runs before the other and vice versa. Note that we generate these orders per test class and do not worry about inter-class test pairs [31].

OD-Checker reports all tests that have failed at least once in any of the orders it runs the tests in. In additional to normal test failures, we also insert a long timeout on each test, where a test that reaches the timeout value also counts as a failure. We insert timeouts on each test because we noticed cases where the transformations can lead to tests to run indefinitely, due to some state not being reset between tests. We want to report such cases as failed tests as well, for the subsequent steps of TestBoost.

3.3 Injector

If OD-Checker finds tests that fail, we use Injector to transform the test class further as to prevent their failures. Since the tests fail due to test fixtures no longer running and resetting state in-between each test, Injector injects explicit calls to test fixtures where needed.

```
1  # testclasses: the transformed test classes to transform
       further
2  # global_failedtests: the tests that failed in some order
3  def Injector(testclasses, global_failedtests):
4    for tc in testclasses:
5      tc_failedtests = set()
6      for t in global_failedtests:
7        if t in tests(tc):
8          tc_failedtests.add(t)
9
10     failedtests = list(tc_failedtests)
11     before_fixtures, after_fixtures = find_fixtures(tc)
12     polluter_victim_pairs = find_polluter_victims(tc,
            failedtests)
13     sorted_polluters_victims = sort_polluters(
            polluter_victim_pairs)
14     sorted_victims_polluters = sort_victims(
            polluter_victim_pairs)
15     while len(failedtests) > 0:
16       num_top_victims = 0, num_top_polluters = 0
17       if len(sorted_polluters_victims) > 0:
18         num_top_victims = len(sorted_polluters_victims
              [0][1])
19       if len(sorted_victims_polluters) > 0:
20         num_top_polluters = len(sorted_victims_polluters
              [0][1])
21       if num_top_victims > 0 || num_top_polluters > 0:
22         if num_top_victims >= num_top_polluters:
23           victims = sorted_polluters_victims[0][1]
24           polluter = sorted_polluters_victims.pop()[0]
25           # three modes, only inject to the polluter;
26           # only inject to the victims;
27           # inject both to the polluter and victims
28           add_fixture_calls(polluter, after_fixtures,
                victims, before_fixtures)
29           pairfailedtests = run_every_pair(polluter,
                victims)
30           if len(pairfailedtests) == 0:
31             continue
32           else:
33             return FAIL
34         else:
35           polluters = sorted_victims_polluters[0][1]
36           victim = sorted_victims_polluters.pop()[0]
37           # three modes, only inject to the victim;
38           # only inject to the polluters;
39           # inject both to the polluters and victim
40           add_fixture_calls(polluters, after_fixtures,
                victim, before_fixtures)
41           pairfailedtests = run_every_pair(polluters,
                victim)
42           if len(pairfailedtests) == 0:
43             continue
44           else:
45             return FAIL
46       else:
47         failedtests = ODChecker(tc)
48         if len(failedtests) > 0:
49           if len(failedtests & tc_failedtests) > 0:
50             return FAIL
51           tc_failedtests |= failedtests
52           polluter_victim_pairs = find_polluter_victims(
                tc, failedtests)
53           sorted_polluters_victims = sort_polluters(
                polluter_victim_pairs)
54           sorted_victims_polluters = sort_victims(
                polluter_victim_pairs)
55
56   return SUCCESS
```

Figure 5: Injector pseudocode.

Figure 5 shows pseudocode for the Injector. Injector first finds what are the corresponding test fixtures for the test class (Line 11). Note that these test fixtures may be declared in a superclass of the failed test. As a means to identify more precisely where to inject calls to test fixtures, Injector first identifies the relevant tests for making the OD test fail. Using terminology by Shi et al. [50], we treat the OD test as a victim that fails when run after another test, a polluter, "pollutes" their shared state[1]. We find all polluters for each victim (Line 12) by running each test before the victim one-by-one with the victim to see whether victim fails [28]. Injector creates a set of all pairs of polluter/victim, and a single polluter may pollute multiple victims or a single victim may have different polluters.

Intuitively, for a victim, if there is a test fixture call at its beginning or there is a test fixture call at the end of each of its polluters, then the victim should pass in any order. The goal is to minimize the number of calls to test fixtures, e.g., if a single test is the polluter for multiple different victims, it would be more efficient to inject a test fixture call at the end of that polluter. Injector injects calls to the teardown test fixture(s) at the end of a polluter, while it injects calls to the setup test fixture(s) at the beginning of a victim. These transformations emulate normal execution involving test fixtures, as to reset shared state between the pair of tests. As long as not all tests need such transformations, we can still have faster runtime due to not having to invoke the test fixtures as often as before.

To figure out the best places to inject test fixture calls, Injector sorts the set of polluter/victim pairs in two ways: one that sorts polluters based on number of victims they form pairs with, and the other that sorts victims based on number of polluters they form pairs with (Lines 13-14). Then, Injector iteratively tries to "fix" each victim by going through the pairs and determining where to inject calls to test fixtures. Given the two sorted lists of pairs, Injector determines whether the top polluter has more victims or the top victim has more polluters, and it will inject test fixture calls to whichever one has more. Injector uses the function `add_fixture_calls` (Line 28) to modify these tests, which it does in one of three ways. If the polluter is chosen (Line 22), `add_fixture_calls` tries to inject the `@After`/`@AfterEach` test fixtures to the end of that polluter, and then it runs the transformed polluter alongside each victim. If this approach cannot make the victims pass, `add_fixture_calls` will revert the last change and inject `@Before`/`@BeforeEach` test fixture calls to the beginning of all the victims. If the victims still fail, `add_fixture_calls` will transform both polluter and all victims together, essentially applying the first two approaches to all relevant tests. `add_fixture_calls` performs a similar transformation if the victim is chosen (Line 40), but it attempts to inject test fixture calls to the victim first before moving on to the polluters. If the victim still does not pass, Injector stops and is unsuccessful at transforming the test class.

After transforming the tests, we run the tests through OD-Checker again to check whether they all pass in any order (Line 47). If there are failing tests seen previously, we are unsuccessful at transforming the test class (Line 49). If there are only newly failing tests, we rerun the main logic of Injector again on these newly failing tests. Injector is successful if all tests pass when run in any order.

[1] We only handle victim tests given that they are the most prominent type of OD tests.

As developers maintain the test class for their future changes, if they add more tests or change existing ones, they can use existing techniques for OD test detection [25, 31, 32], and if there are OD tests, they can apply TestBoost to address this issue.

4 METHODOLOGY

We answer the following research questions:

- RQ1: How effective is TestBoost at transforming test fixtures?
- RQ2: How much reduction in runtime per test class does TestBoost achieve after transforming test fixtures?
- RQ3: How much reduction in overall test suite runtime does TestBoost achieve after transforming test fixtures?
- RQ4: How much fault-detection capability is reduced after transforming test fixtures?
- RQ5: How do developers react to transforming test fixtures?

We answer RQ1 to evaluate how well TestBoost can transform test classes and test fixtures. We answer RQ2 and RQ3 to evaluate how much faster are the test classes and overall test suite, respectively, after transformation. We answer RQ4 to check potential test quality loss after transformation. Finally, we answer RQ5 to see whether developers are receptive to such transformations.

4.1 Dataset

Our dataset consists of open-source projects used in prior research on software testing [30, 31, 40, 51]. These prior works generally proposed techniques to speed up testing, so their subjects would be a good fit for our goals. We implement TestBoost for Maven, so we start with the 91 Maven projects from those prior works.

We further filter the test classes to contain those that take at least one second to run, contain test fixtures, and contain more than one test (there is no point in transforming test fixtures to run once per test class if there is just one test). We obtain 921 test classes to transform. While TestBoost can transform superclasses of a target test class, if those superclasses are outside of the current test suite, e.g., from a third-party library, TestBoost cannot transform the test files corresponding to those test classes. We therefore remove 169 test classes that inherit from classes outside of its test suite. After running the tests in these test classes in different orders, we exclude 55 test classes that already contain some OD tests beforehand, since TestBoost assumes tests initially can pass in any order as to determine whether it can successfully transform a test class.

Table 1 shows the characteristics of the projects and tests we use in our evaluation. Column "ID" shows a project ID that we use to refer to the project in future tables. Column "Project" shows the project GitHub user/repository. Column "# Mod." shows the number of modules we use from that project[2]. Column "# TC" shows the number of test classes and column "# TM" shows the number of tests within those test classes. Column "Avg. TC runtime" shows the average runtime in seconds of each test class per project. Finally, Column "Avg. Module runtime" shows the average runtime in seconds of the entire test suite in each module per project. The final row shows the total number of modules, test classes, and tests we use in our evaluation, and it shows the average runtime of all test classes and module test suites we use in our evaluation. Overall, we evaluate on 697 test classes with 38430 tests, spread across 135 modules from 34 projects. The average test class and test suite runtimes are 11.11 seconds, and 112.06 seconds, respectively.

4.2 Metrics

To answer RQ1, we measure how many test classes TestBoost can transform successfully. We also evaluate the rate of test fixture injection (Section 3.3), calculated as:

$$\frac{Vic * Setup + Pol * Teardown}{Tests * (Setup + Teardown)}$$

where Vic and Pol are the the number of victims and number of polluters, respectively, where we inject calls to test fixtures. $Setup$ and $Teardown$ are the number of setup and teardown test fixtures, respectively, in the test class, and $Tests$ is the number of tests. The denominator is the total number of test fixture calls that would happen normally (all test fixture methods get called for every test), and the numerator represents the more targeted number of test fixture calls made. A smaller injection rate is desirable.

To answer RQ2, we measure the test class runtime before and after transformation and compute runtime reduction as:

$$\frac{Runtime(T_b) - Runtime(T_a)}{Runtime(T_b)}$$

where $Runtime()$ computes the test class runtime, averaged across 10 runs, for the test class T_b before transformation and T_a after transformation. A higher reduction is better, indicating a greater decrease in test runtime. Note that the percentage may be negative, indicating increased test runtime. We perform a Wilcoxon signed-rank test to check for statistically significant difference in runtime.

To answer RQ3, for each module, we replace all test classes in the test suite with the transformed test classes with positive statistically significant runtime reduction. We ignore any test classes with insignificant or negative reduction in runtime, because a developer would not use such transformed test classes. We run the entire test suite of the module before and after including the transformed test classes and measure the reduction in test suite runtime. We perform a Wilcoxon signed-rank test to measure significant differences.

To answer RQ4, we use JaCoCo [6] to measure line coverage of test classes as a proxy for fault-detection capability. We collect the set of lines covered before/after transformation and check which lines that were covered before are no longer covered after, measuring percentage loss in coverage as:

$$\frac{|Cov(T_b) \setminus Cov(T_a)|}{|Cov(T_b)|}$$

where $Cov()$ computes the set of lines covered by the test class.

We also use PIT [7] to conduct mutation testing analysis [22] to measure fault-detection capability. We run PIT on the test class before/after transformation to collect the set of killed mutants, measuring loss in mutants killed as:

$$\frac{|Kill(T_b) \cap Surv(T_a)|}{|Kill(T_b)|}$$

where $Kill()$ computes the set of mutants killed by the test class and $Surv()$ computes the mutants that survived. We only consider as "killed" the mutants PIT explicitly marks as `KILLED`, excluding mutants that timed out or had memory/run error, which can be

[2] A Maven project can have multiple modules; each module has its own test suite

Table 1: Characteristics of projects in dataset.

ID	Project	# Mod.	# Test Classes	# Test Methods	Avg. Test Class Runtime (s)	Avg. Mod. Runtime (s)
P1	Activiti/Activiti	4	4	15	8.64	45.88
P2	AdoptOpenJDK/jitwatch	1	1	13	9.02	18.42
P3	Graylog2/graylog2-server	1	3	17	2.21	100.89
P4	apache/commons-codec	1	1	117	3.54	10.75
P5	apache/commons-configuration	1	2	87	3.95	28.33
P6	apache/commons-dbcp	1	8	346	4.89	95.68
P7	apache/commons-io	1	3	25	2.59	72.76
P8	apache/commons-lang	1	1	10	5.47	69.15
P9	apache/commons-math	2	2	93	2.59	13.68
P10	apache/commons-pool	1	4	183	71.70	377.84
P11	apache/dubbo	16	29	227	5.88	24.06
P12	apache/flink	11	32	445	5.90	259.71
P13	apache/hadoop	29	401	4146	17.97	1014.68
P14	apache/incubator-skywalking	1	1	2	10.40	10.34
P15	apache/rocketmq	6	21	196	17.72	70.50
P16	apache/storm	2	5	21	9.79	77.74
P17	brettwooldridge/HikariCP	1	6	47	18.07	220.51
P18	dropwizard/dropwizard	9	13	124	3.05	15.60
P19	druid-io/druid	17	100	31453	9.91	430.39
P20	eclipse/eclipse-collections	1	1	33	1.93	88.96
P21	elasticjob/elastic-job-lite	2	3	22	1.82	10.55
P22	graphhopper/graphhopper	1	3	308	4.22	117.99
P23	igniterealtime/Openfire	1	8	191	2.04	72.44
P24	iluwatar/java-design-patterns	4	4	15	6.76	9.52
P25	ktuukkan/marine-api	1	1	16	1.89	2.37
P26	languagetool-org/languagetool	6	11	114	9.77	69.02
P27	spring-projects/spring-ws	1	3	10	7.52	19.39
P28	srt/asterisk-java	1	1	22	4.45	35.03
P29	undertow-io/undertow	2	7	45	8.35	210.36
P30	vmware/admiral	1	1	3	2.03	14.69
P31	wikidata/wikidata-toolkit	1	1	26	1.30	2.08
P32	wildfly/wildfly	2	5	13	10.39	40.68
P33	wso2/carbon-apimgt	2	3	11	97.16	145.98
P34	zalando/riptide	3	8	34	4.71	14.04
	Avg./Total	**135**	**697**	**38430**	**11.11**	**112.06**

due to various nondeterministic factors, e.g., a machine slowdown. Conversely, "survived" mutants are those PIT marks as SURVIVED or NO_COVERAGE, so they definitely are not killed.

To answer RQ5, we send one pull request for each project that has at least one transformed test class that could statistically significantly lower the module's test suite runtime. We send a pull request with just one test class per project as to not overwhelm them with too many changes that they may not want.

4.3 Running Environment

We run all our experiments in a Docker container built from an Ubuntu 20.04 Docker image, using JDK 17 and Maven 3.8.3. We limit the container to use 2 CPUs and 8GB of RAM, simulating the environment commonly used in continuous integration services [3, 5], which is where we expect most developers to run their tests.

5 EVALUATION

5.1 RQ1: Effectiveness of Transformations

Table 2 shows the results of applying TestBoost on our dataset of test classes. Column "# Success" shows the number of test classes on which TestBoost could successfully transform the test fixtures with the tests still passing. In total, TestBoost successfully transforms 500 out of the 697 test classes, i.e., 71.74% of them. TestBoost on average uses 2646.76 seconds to transform a test class (running all three components of TestBoost). The majority of this time is used in OD-Checker as well as Injector searching for polluters, which makes sense given the large number of reruns needed by these components. Fortunately, transformation only needs to be done once, as the transformed test class can be re-used in the future.

We inspect further the reasons for why TestBoost cannot successfully transform the 197 test classes. Eight test classes contain flaky

Table 2: Successfully transformed test classes characteristics.

ID	# Success	% Injections
P1	1	50.00
P2	1	0.00
P4	1	5.56
P5	2	32.64
P11	27	10.69
P12	24	13.69
P13	293	25.71
P15	13	18.83
P17	6	20.14
P18	10	18.38
P19	59	28.43
P21	3	47.35
P23	8	22.54
P26	10	0.00
P27	3	0.00
P29	7	14.29
P33	3	66.67
P34	8	8.33
Avg./Total	**500**	**24.25**

tests [35] before transformation, meaning they nondeterministically pass/fail when run multiple times. The failures after transformation may not be due to the transformation itself. For three test classes, we find that the tests became flaky after the transformation. These tests indeed had logic referring to shared state, so the transformation likely increased the chance of flaky failure, though the tests do not consistently pass/fail in specific orders as prior work defined order-dependent tests [27, 50]. TestBoost can report whether tests have such flaky behavior, allowing developers to decide to use the transformed test classes. We inspected further one case, where the test fixture helps change the status of a node in a cluster to active, allowing the test to write to it. However, activation may take some time and writing to an inactive node raises errors. While this test is flaky and can fail even before transformation, by not calling the test fixture as often, the chance of failure ends up increasing.

We cannot transform 10 test classes due to specific properties of the Java class that we currently do not handle, e.g., abstract test classes with abstract methods or type parameters. Two test classes skip to run some test methods after the transformation that were not skipped before. TestBoost fails to transform 73 test classes successfully due to issues related to constructors (we do not transform non-default constructors), or the test fixtures specifically rely on being run per test, e.g., getting the current test method name, which is not possible for a static test fixture. Two test classes do not compile due to removing the `final` keyword (Section 3.1).

Finally, 99 test classes do not pass even though we inject calls to test fixtures. Some examples include being unable to change a `@Rule` field into `@ClassRule`, or in some cases the test where we inject a test fixture call runs first in some order, making the test fixture executed twice in a row and breaking intended functionality.

Table 2 also shows under Column "% Injections" the average injection rate per test class (Section 4.2), with average 24.25% per test class. For the most part, TestBoost does not need to inject calls to test fixtures to the test classes, indicating that tests can still pass even if the test fixtures are not run before/after all tests in the test class. In fact, 237 test classes can run all tests successfully without needing any injected calls. However, several test classes with many tests require injected calls, with a maximum number of 51 methods needing injected calls in a single test class from project P5.

Table 3: Reduction in runtime per test class.

ID	# Signf.	Red. signf. %	# Pos. signf.	Pos. Red. %
P1	1	2.44	1	2.44
P2	1	90.53	1	90.53
P4	1	12.73	1	12.73
P5	1	26.99	1	26.99
P11	16	14.10	16	14.10
P12	10	17.73	10	17.73
P13	105	14.71	87	19.06
P15	7	10.60	6	11.85
P17	1	-13.91	0	0.00
P18	6	13.13	5	23.67
P19	23	17.47	22	17.63
P21	2	-16.07	1	48.39
P23	4	-8.00	3	31.30
P26	7	11.96	7	11.96
P27	3	83.23	3	83.23
P29	3	17.44	2	60.48
P33	1	7.29	1	7.29
P34	2	3.26	2	3.26
Percent. Avg.		**15.24**		**20.01**
Runtime Avg.		**15.32**		**19.16**
Avg./Total	**194**	**16.98**	**169**	**28.39**

RQ1 Summary: TestBoost can successfully transform most test classes from our dataset, transforming 500 out of the 697 total test classes in our dataset. Further, 237 of the transformed test classes still pass even without needing to inject any calls to test fixtures for any test methods, and the average injection rate is 24.25%.

5.2 RQ2: Reduction in Test Class Runtime

Table 3 shows the results of comparing the runtime before and after transforming test classes. Column "# Signf." shows the number of test classes where runtime is statistically significantly different, $p < 0.05$. Overall, 194 test classes out of 500 successfully transformed test classes have significant runtime difference.

However, the significant differences are not all positive. We see several cases where there is no reduction in test runtime but rather an increase. Column "Red. signf. %" shows the average reduction in test runtime after the transformation per test class per project. A negative percentage indicates an increase in test runtime. Overall, though, we see that on average there is still a positive reduction in runtime. The average reduction per test class across all projects is 15.24%. If we weight by runtime (compute reduction in terms of the sum of runtime across all test classes, giving more weight to test classes with higher runtime), the average reduction is 15.32%. The two averages are about the same, suggesting that the reduction

Table 4: Reduction in runtime per module test suite.

ID	# Mod.	# Pos. signf.	Pos. Red. %
P1	1	0	0.00
P2	1	1	81.28
P4	1	1	5.49
P5	1	1	4.88
P11	10	7	6.20
P12	5	1	6.71
P13	14	4	4.26
P15	3	2	6.58
P18	4	4	11.48
P19	6	3	8.65
P21	1	1	3.34
P23	1	1	4.66
P26	4	1	29.74
P27	1	1	85.35
P29	2	1	2.85
P33	1	1	12.14
P34	2	0	0.00
Percent. Avg.			**12.73**
Runtime Avg.			**5.17**
Avg./Total	**58**	**30**	**18.24**

in runtime is not that dependent on individual test class runtimes. Finally, the average reduction per project is 16.98%.

In general, TestBoost provides little runtime reduction in test classes with few test methods, since test fixtures are not run that often in the first place, or when the test fixtures do not run very long compared to the test methods. When measuring R^2 correlation between number of test methods per test class and runtime reduction, we find a positive correlation (0.02), so more test methods means bigger reduction, but the correlation is small, suggesting the main contributing factor is likely the runtime of a test fixture relative to test methods. Further, we inspect some of the more extreme cases, such as a test class from project P23 with an increased runtime (-189.43% "reduction"). We observe two threads running per test, and the test fixtures try to release the locks of these two threads. If we transform the test class, the test methods wait for the threads to release the locks, making the runtime go up.

Column "# Pos. signf." in Table 3 shows the number of test classes with positive, statistically significant reduction in test runtime. Overall, 169 test classes' runtimes are significantly reduced, so most test classes have a positive reduction in runtime. A developer should only use these 169 transformed test classes. If we consider the reduction in test class runtime for just these positive cases (column "Pos. Red. %"), the average reduction per test class is 20.01%. When weighted by runtime, the average reduction is 19.16%. The average reduction per project is 28.39%.

RQ2 Summary: 169 test classes have significant reduced runtime after transformation, leading to an average runtime reduction of 28.39% per project.

5.3 RQ3: Reduction in Test Suite Runtime

Table 4 shows the results of the reduction in test suite runtime after using all transformed test classes that provide positive significantly reduced runtimes in that test suite. The column "# Mod." shows the number of modules per project where we have a test class with a positive significantly reduced runtime, column "# Pos. signf." shows the number of modules where the reduction is significant and positive, and column "Pos. Red. %" shows the average reduction percentage for those positive significant cases. We observe only one module with significant negative reduction.

Overall, 58 modules have transformed test classes with positive significant reduction in runtime, with 30 modules having a significant reduction in test suite runtime. The average reduction in test suite runtime for these modules is 12.73%. When weighted by module runtime, the average reduction is 5.17%. This runtime weighted average is much lower, suggesting that the longer-running module test suites have less reduction. The reason for this effect may be that these test suites have many tests, so transforming just a few of the test classes may not change the overall test suite runtime as much. Finally, the average reduction per project is 18.24%.

We inspect some of the more extreme cases where the reduction in test suite runtime is very large. For example, for project P2, the transformed test class originally counts for a large portion of the overall test suite runtime in the module. Further, the test class has reduction in runtime of 90.53%, which we find is due to the `@Before` test fixture performing some expensive operations to compile some classes that runs before every test. Changing this test fixture to `@BeforeClass` greatly reduces the runtime, which in turn contributes a lot to the reduction in test suite runtime.

RQ3 Summary: TestBoost's transformed test classes can provide a positive significant difference in runtime for the entire test suite in the module for 30 out of the 58 modules, providing an average reduction of 18.24% in runtime.

5.4 RQ4: Fault-Detection Capability

We collect the coverage information for 168 test classes and 56 modules. We lose one test class from P33 where the project explicitly disables JaCoCo due to integration issues, so we also cannot collect coverage. Table 5 shows loss in coverage due to transformation. Column "# Test Classes" shows the number of test classes that have loss in coverage from before. Column "% Lines (TC)" shows the percentage coverage decrease per test class. Column "% Lines (TS)" shows the percentage coverage decrease on the whole test suite after including all transformed test classes. Overall, 54 test classes lose coverage of lines covered before transformation, with average percentage decrease of 0.09% per test class and average percentage decrease of coverage per test suite across all projects is 0.05% per test suite. Loss of coverage is overall <1%. We also observe gain in coverage per test class, with 62 test classes covering additional lines, average 0.44% per test class and 0.07% per test suite. Interestingly, `undertow-io/undertow` has decreased coverage at the test suite level without any loss in coverage per test class. We find it due to flakiness in coverage of other test classes we did not transform [21].

In measuring mutants killed, we can only collect such information for 101 test classes and 19 modules; PIT could not run properly for the remaining test classes and modules due to issues with how

Table 5: Decrease in lines covered.

ID	# Test Classes	% Lines (TC)	% Lines (TS)
P1	1	0.29	0.16
P2	0	0.00	0.00
P4	0	0.00	0.00
P5	0	0.00	0.00
P11	1	0.02	0.00
P12	1	0.01	0.02
P13	47	0.09	0.18
P15	1	0.51	0.02
P18	0	0.00	0.00
P19	3	0.16	0.00
P21	0	0.00	0.00
P23	0	0.00	0.00
P26	0	0.00	0.00
P27	0	0.00	0.00
P29	0	0.00	0.03
P34	0	0.00	0.00
Avg./Total	**54**	**0.09**	**0.05**

Table 6: Decrease in mutants killed.

ID	# Test Classes	% Muts (TC)	% Muts (TS)
P1	1	2.21	-
P2	0	0.00	-
P4	0	0.00	0.00
P5	0	0.00	0.00
P11	1	2.50	0.00
P12	5	0.86	-
P13	15	1.62	0.00
P15	5	7.73	0.23
P18	0	0.00	0.00
P19	3	0.38	-
P21	0	0.00	0.00
P23	1	9.43	0.04
P26	0	0.00	0.01
P27	0	0.00	0.00
P29	0	0.00	-
P33	0	0.00	-
P34	0	0.00	0.00
Avg./Total	**31**	**1.67**	**0.02**

PIT integrates with those projects, even before transformation. Table 6 shows the change in mutants killed after transformation. Column "# Test Classes" shows the number of test classes that do not kill some mutants originally killed before transformation. Column "% Muts (TC)" shows the the percentage decrease of mutants killed per test class. Column "% Muts (TS)" shows the percentage decrease of mutants killed per test suite. Overall, 31 of the test classes miss to kill some mutants killed before. The average percentage decrease of mutants killed per test class across all projects is 1.67%. The average percentage decrease of mutants killed per test suite is 0.02%. We notice test classes with additional killed mutants after transformation, with 22 test classes killing additional mutants, with average percentage increase of 1.00% killed mutants per test class. We also notice loss in mutants killed at the test suite level when there is none at the test class level due to flakiness in coverage [47].

RQ4 Summary: 54 test classes lose covered lines after transformation, with average percentage decrease of coverage per test class of 0.09%. 31 test classes have fail to kill mutants killed before, with average percentage decrease of mutants killed per test class of 1.67%. Both losses in coverage and killed mutants is small, and there is even some gain in other lines covered and mutants killed.

5.5 RQ5: Pull Requests

We have so far sent 15 pull requests to 14 projects. So far, we have 9 pull requests merged by the developers, 4 rejected, while 2 are still pending. We did not send a pull request to P23, because we noticed the developers had since refactored the test class by applying a similar approach to move the time-consuming part of the `@Before` method into a `@BeforeClass` method, illustrating the importance of TestBoost in automating tasks a developer wants to perform.

For those 2 pending pull requests, while they have not made any official decision on the changes, we have been in discussion with developers, receiving valuable insights. The developers of P12 believe their tests are unstable, and they worry that these changes may make them more vulnerable and flaky in the future. The overall one second reduction in runtime that our transformation would bring would not be good enough for them to consider the changes. Maintainers from P13 have not provided feedback yet. Concerning the 4 rejected pull requests, for one from P19, developers closed our pull request, because they did not want to set every field involved in the `@Before` method to be static, where only certain targeted fields should be moved to `@BeforeClass`. Currently, we transform as much as necessary to make all test fixtures static methods. The developers made their own changes to test fixtures, fundamentally using a similar approach but moving only targeted parts of code into the static test fixtures. Our changes to tests from P5 and P4 require injecting calls to test fixtures in tests, and developers say it is not worth making the tests more convoluted, especially given the small reduction in runtime. We did not receive any feedback for the remaining rejected pull request from P26 before it got closed.

Based on this feedback, we see that developers care about changes that can result in substantially high amounts of runtime reduction. In terms of absolute runtime, test classes that only take around one second to run may not be worth it for developers to consider improving. However, we have some accepted pull requests where the test class takes only 1-2 seconds to run. We also observe that additional changes such as injecting calls to test fixtures in targeted methods may be too convoluted and irregular that they would not consider unless runtime reductions are substantial enough.

We also measure the loss in coverage and mutants killed for accepted transformed test classes, finding only one test class with loss in coverage (3.04% decrease) and two test classes with loss in mutants killed (average 1.64% decrease). Once again, most test classes did not have any loss in coverage nor mutants killed, and when there are losses the reduction is low. Despite some loss, developers still accepted the pull requests, suggesting the loss is acceptable.

RQ5 Summary: We submitted 15 pull requests to developers to transform their test classes, with 9 merged and 2 pending.

6 THREATS TO VALIDITY

Our implementation of TestBoost may have bugs. We reviewed the code and logic to confirm that it indeed transforms test classes as we need. Further, we check that the tests indeed pass after the transformation; our evaluation consists only of tests that pass.

The runtime reductions we observe may not generalize to all projects. We choose a diverse set of projects from prior work on testing. When we run the tests to collect runtimes, we rerun them multiple times before and after the transformations to collect an average runtime, mitigating the effects of natural variance in runtime. We also conduct a statistics test to measure for significance in the differences for all runtime comparisons. While we develop TestBoost specifically for JUnit and Java, we believe the general idea is applicable to projects using other testing frameworks that include have some concept of test fixtures that run in-between tests.

By changing test fixtures, we may be changing test functionality. We use the test results as an indication of behavior, aiming to preserve the same passing tests outcomes after the transformations. The successful cases are when the transformations result in tests passing, and we measure runtime reductions only for successful cases. We submit transformed test classes as pull requests to developers, so they can also check our changes.

7 RELATED WORK

Regression Testing Techniques. Common regression testing techniques are test-suite reduction (TSR), regression test selection (RTS), or test-case prioritization (TCP) [55]. TSR creates a reduced test suite by removing from the test suite the redundant tests, computed w.r.t. some coverage heuristics, e.g., removing tests that cover the same lines [9, 10, 18, 24, 37, 44, 48, 58]. Prior work evaluated loss in fault-detection capability using mutants, where some loss may be acceptable. However, Shi et al. found that those reduced test suites are not as effective at detecting real faults on future versions of code [49], showing the risk of removing tests. In contrast, we speed up testing without removing tests. We are similar to work by Vahabzadeh et al. [52] that removed redundant statements in tests, except we transform test fixtures to run less frequently.

RTS analyzes the changes between code versions and only runs tests affected by changes. Prior RTS techniques mostly focused on static or dynamic analysis to map changes to affected tests [14, 16, 19, 20, 29, 41, 43], with the goal of selecting fewer tests to run faster. However, there is still risk that RTS does not select the correct tests, and Zhu et al. developed RTSCheck to evaluate RTS techniques, finding faults in RTS tools [60]. Recent work in RTS leverage machine learning techniques to predict which tests could fail after the changes [13, 38, 56]. These techniques may make wrong predictions, thereby also risking to miss to select necessary tests. Our approach speeds up testing while still running all tests. One could also combine RTS with TestBoost, to select to run a subset of already transformed test classes. Note that our transformations ensure that test fixtures will run once before/after the test class, ensuring isolation between them. Prior work found RTS works best at the test class level for Java [16, 29], making them compatible.

TCP runs tests in an optimized order so tests more likely to detect faults run earlier [36, 55]. TCP techniques use heuristics like code coverage [45], diversity of code covered between tests [23], or information retrieval [42, 46] to rank tests. TCP runs all tests, so there is generally no runtime reduction. We also run all tests, but we transform test classes to run faster with fewer test fixture calls.

Flaky Tests. Flaky tests can nondeterministically pass or fail when run on the same version of code [11, 35]. One prominent type of flaky tests are order-dependent flaky tests (OD tests), whose outcomes change depending on the order in which they are run [35, 59]. These tests can fail due to relying on shared state modified by other tests, e.g., they rely on another test to set up the shared state or fail because another test modifies the shared state without cleaning. Test fixtures may be used to setup that state or clean modified state between tests, so our approach that transforms test fixtures can lead to tests to become OD tests. Prior work proposed rerunning tests in random orders [25, 53, 59], systematically generating test orders to ensure pairs of tests get covered [31, 54, 59], or tracking shared state between tests [8, 15, 17]. We rely on systematically generating test orders to detect OD tests. Beyond detection, Shi et al. proposed repairing OD tests by looking for "cleaner" tests in the test suite that contains code to reset and clean the shared state [50]. Li et al. followed up with work to generate such cleaners by analyzing shared state and searching for the relevant reset methods [33]. These techniques have a similar effect as generating test fixtures. Conversely, our approach reduces the effects of existing test fixtures, having them not run before/after each test, which is what leads to new OD tests. If needed, we inject calls to test fixtures in the specific OD tests that fail as a more targeted way to reset shared state. Lam et al. proposed partially enforcing dependencies between tests as to mitigate their effects on regression testing techniques [26].

8 CONCLUSIONS

We propose TestBoost, an approach for reducing test runtime by transforming test fixtures. Test classes with test fixtures run them before/after each test, but tests may not always need these test fixtures to run as frequently. TestBoost transforms the test fixtures such that they run only once at the beginning or at the end of the test class, running just once per test class instead of once per test. TestBoost successfully transforms 500 out of 697 test classes, resulting in zero test classes with OD tests. Further, the transformed test classes with positive significant reductions in runtime has average reduction in runtime of 28.39%; using these transformed test classes in the test suite can reduce the overall test suite runtime on average by 18.24%. In the future, we plan to look into more efficient means to inject calls to test fixtures as to reduce extra runtime needed by these extra calls, as well as looking into how to do more fine-grained changes, extracting parts of the code in test fixtures into static test fixtures, allowing a mix of both types of test fixtures. Such fine-grained changes may allow for more test classes to be transformed successfully.

ACKNOWLEDGMENTS

We thank Xingchen Qi for his help in formulating initial ideas of this work and for exploring the use of running tests in random orders to detect newly OD tests. This work is partially supported by the US National Science Foundation under Grant Nos. CCF-2145774 and CCF-2217696, as well as the Jarmon Innovation Fund.

REFERENCES

[1] 2011. Testing at the speed and scale of Google. http://google-engtools.blogspot.com/2011/06/testing-at-speed-and-scale-of-google.html.

[2] 2019. JavaParser. http://javaparser.org.

[3] 2023. GitHub Actions. https://github.com/features/actions.

[4] 2023. Reducing Test Runtime by Transforming Test Fixtures. https://sites.google.com/view/transforming-test-fixtures.

[5] 2023. Travis-CI. https://travis-ci.org.

[6] 2024. JaCoCo Java Code Coverage Library. https://www.eclemma.org/jacoco/.

[7] 2024. PIT Mutation Testing. http://pitest.org.

[8] Jonathan Bell, Gail Kaiser, Eric Melski, and Mohan Dattatreya. 2015. Efficient Dependency Detection for Safe Java Test Acceleration. In *International Symposium on Foundations of Software Engineering*. 770–781.

[9] T. Y. Chen and M. F. Lau. 1998. A new heuristic for test suite reduction. *Journal of Information and Software Technology* 40, 5-6 (1998), 347–354.

[10] T. Y. Chen and M. F. Lau. 1998. A simulation study on some heuristics for test suite reduction. *Journal of Information and Software Technology* 40, 13 (1998), 777–787.

[11] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. 2019. Understanding Flaky Tests: The Developer's Perspective. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 830–840.

[12] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for Improving Regression Testing in Continuous Integration Development Environments. In *International Symposium on Foundations of Software Engineering*. 235–245.

[13] Daniel Elsner, Florian Hauer, Alexander Pretschner, and Silke Reimer. 2021. Empirically Evaluating Readily Available Information for Regression Test Optimization in Continuous Integration. In *International Symposium on Software Testing and Analysis*. 491–504.

[14] Emelie Engström, Mats Skoglund, and Per Runeson. 2008. Empirical evaluations of regression test selection techniques: A systematic review. In *International Symposium on Empirical Software Engineering and Measurement*. 22–31.

[15] Alessio Gambi, Jonathan Bell, and Andreas Zeller. 2018. Practical test dependency detection. In *International Conference on Software Testing, Verification, and Validation*. 1–11.

[16] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical Regression Test Selection with Dynamic File Dependencies. In *International Symposium on Software Testing and Analysis*. 211–222.

[17] Alex Gyori, August Shi, Farah Hariri, and Darko Marinov. 2015. Reliable testing: Detecting state-polluting tests to prevent test dependency. In *International Symposium on Software Testing and Analysis*. 223–233.

[18] Dan Hao, Lu Zhang, Xingxia Wu, Hong Mei, and Gregg Rothermel. 2012. On-demand test suite reduction. In *International Conference on Software Engineering*. 738–748.

[19] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S. Alexander Spoon, and Ashish Gujarathi. 2001. Regression Test Selection for Java Software. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 312–326.

[20] Mary Jean Harrold, David Rosenblum, Gregg Rothermel, and Elaine Weyuker. 2001. Empirical studies of a prediction model for regression test selection. *IEEE Transactions on Software Engineering* 27, 3 (2001), 248–263.

[21] Michael Hilton, Jonathan Bell, and Darko Marinov. 2018. A Large-Scale, Longitudinal Study of Test Coverage Evolution. In *International Conference on Automated Software Engineering*. 53–63.

[22] 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.

[23] Bo Jiang, Zhenyu Zhang, Wing Kwong Chan, and T. H. Tse. 2009. Adaptive random test case prioritization. In *International Conference on Automated Software Engineering*. 233–244.

[24] James A. Jones and Mary Jean Harrold. 2001. Test-suite reduction and prioritization for modified condition/decision coverage. In *International Conference on Software Maintenance*. 92–102.

[25] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. 2019. iDFlakies: A framework for detecting and partially classifying flaky tests. In *International Conference on Software Testing, Verification, and Validation*. 312–322.

[26] Wing Lam, August Shi, Reed Oei, Sai Zhang, Michael D. Ernst, and Tao Xie. 2020. Dependent-Test-Aware Regression Testing Techniques. In *International Symposium on Software Testing and Analysis*. 298–311.

[27] Wing Lam, Stefan Winter, Angello Astorga, Victoria Stodden, and Darko Marinov. 2020. Understanding Reproducibility and Characteristics of Flaky Tests Through Test Reruns in Java Projects. In *International Symposium on Software Reliability Engineering*. 403–413.

[28] Wing Lam, Stefan Winter, Anjiang Wei, Tao Xie, Darko Marinov, and Jonathan Bell. 2020. A large-scale longitudinal study of flaky tests. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–29.

[29] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. 2016. An Extensive Study of Static Regression Test Selection in Modern Software Evolution. In *International Symposium on Foundations of Software Engineering*. 583–594.

[30] Owolabi Legunsen, August Shi, and Darko Marinov. 2017. STARTS: STAtic regression test selection. In *International Conference on Automated Software Engineering*. IEEE, 949–954.

[31] Chengpeng Li, M Mahdi Khosravi, Wing Lam, and August Shi. 2023. Systematically Producing Test Orders to Detect Order-Dependent Flaky Tests. In *International Symposium on Software Testing and Analysis*. 627–638.

[32] Chengpeng Li and August Shi. 2022. Evolution-aware detection of order-dependent flaky tests. In *International Symposium on Software Testing and Analysis*. 114–125.

[33] Chengpeng Li, Chenguang Zhu, Wenxi Wang, and August Shi. 2022. Repairing Order-Dependent Flaky Tests via Test Generation. In *International Conference on Software Engineering*. 1881–1892.

[34] Yu Liu, Jiyang Zhang, Pengyu Nie, Milos Gligoric, and Owolabi Legunsen. 2023. More Precise Regression Test Selection via Reasoning about Semantics-Modifying Changes. In *International Symposium on Software Testing and Analysis*. 664–676.

[35] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An Empirical Analysis of Flaky Tests. In *International Symposium on Foundations of Software Engineering*. 643–653.

[36] Qi Luo, Kevin Moran, and Denys Poshyvanyk. 2016. A large-scale empirical comparison of static and dynamic test case prioritization techniques. In *International Symposium on Foundations of Software Engineering*. 559–570.

[37] Xue-ying Ma, Bin-kui Sheng, and Cheng-qing Ye. 2005. Test-suite reduction using genetic algorithm. In *International Conference on Advanced Parallel Processing Technologies*. 253–262.

[38] Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. 2019. Predictive test selection. In *International Conference on Software Engineering, Software Engineering in Practice*. 91–100.

[39] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-scale continuous testing. In *International Conference on Software Engineering, Software Engineering in Practice*. 233–242.

[40] Pengyu Nie, Ahmet Celik, Matthew Coley, Aleksandar Milicevic, Jonathan Bell, and Milos Gligoric. 2020. Debugging the performance of Maven's test isolation: Experience report. In *International Symposium on Software Testing and Analysis*. 249–259.

[41] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. 2004. Scaling regression testing to large software systems. In *International Symposium on Foundations of Software Engineering*. 241–251.

[42] Qianyang Peng, August Shi, and Lingming Zhang. 2020. Empirically Revisiting and Enhancing IR-Based Test-Case Prioritization. In *International Symposium on Software Testing and Analysis*. 324–336.

[43] Gregg Rothermel and Mary Jean Harrold. 1997. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering Methodology* 6, 2 (1997), 173–210.

[44] Gregg Rothermel, Mary Jean Harrold, Jeffery von Ronne, and Christie Hong. 2002. Empirical studies of test-suite reduction. *Journal of Software Testing, Verification and Reliability* 12, 4 (2002), 219–249.

[45] G. Rothermel, R.H. Untch, Chengyun Chu, and M.J. Harrold. 1999. Test case prioritization: an empirical study. In *International Conference on Software Maintenance*. 179–188.

[46] Ripon K. Saha, Lingming Zhang, Sarfraz Khurshid, and Dewayne E. Perry. 2015. An information retrieval approach for regression test prioritization based on program changes. In *International Conference on Software Engineering*. 268–279.

[47] August Shi, Jonathan Bell, , and Darko Marinov. 2019. Mitigating the effects of flaky tests on mutation testing. In *International Symposium on Software Testing and Analysis*. 112–122.

[48] August Shi, Alex Gyori, Milos Gligoric, Andrey Zaytsev, and Darko Marinov. 2014. Balancing trade-offs in test-suite reduction. In *International Symposium on Foundations of Software Engineering*. 246–256.

[49] August Shi, Alex Gyori, Suleman Mahmood, Peiyuan Zhao, and Darko Marinov. 2018. Evaluating Test-Suite Reduction in Real Software Evolution. In *International Symposium on Software Testing and Analysis*. 84–94.

[50] August Shi, Wing Lam, Reed Oei, Tao Xie, and Darko Marinov. 2019. iFixFlakies: A Framework for Automatically Fixing Order-Dependent Flaky Tests. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 545–555.

[51] August Shi, Peiyuan Zhao, and Darko Marinov. 2019. Understanding and Improving Regression Test Selection in Continuous Integration. In *International Symposium on Software Reliability Engineering*. 228–238.

[52] Arash Vahabzadeh, Andrea Stocco, and Ali Mesbah. 2018. Fine-grained test minimization. In *International Conference on Software Engineering*. 210–221.

[53] Ruixin Wang, Yang Chen, and Wing Lam. 2022. IPFlakies: A Framework for Detecting and Fixing Python Order-Dependent Flaky Tests. In *International Conference on Software Engineering (Tool Demonstrations Track)*. 120–124.

[54] Anjiang Wei, Pu Yi, Tao Xie, Darko Marinov, and Wing Lam. 2021. Probabilistic and Systematic Coverage of Consecutive Test-Method Pairs for Detecting Order-Dependent Flaky Tests. In *Tools and Algorithms for the Construction and Analysis*

of Systems. 270–287.

[55] Shin Yoo and Mark Harman. 2012. Regression Testing Minimization, Selection and Prioritization: A Survey. *Journal of Software Testing, Verification and Reliability* 22, 2 (2012), 67–120.

[56] Jiyang Zhang, Yu Liu, Milos Gligoric, Owolabi Legunsen, and August Shi. 2022. Comparing and Combining Analysis-Based and Learning-Based Regression Test Selection. In *ACM/IEEE International Conference on Automation of Software Test*. 17–28.

[57] Lingming Zhang. 2018. Hybrid regression test selection. In *International Conference on Software Engineering*. 199–209.

[58] Lingming Zhang, Darko Marinov, Lu Zhang, and Sarfraz Khurshid. 2011. An empirical study of JUnit test-suite reduction. In *International Symposium on Software Reliability Engineering*. 170–179.

[59] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kıvanç Muşlu, Wing Lam, Michael D. Ernst, and David Notkin. 2014. Empirically revisiting the test independence assumption. In *International Symposium on Software Testing and Analysis*. 385–396.

[60] Chenguang Zhu, Owolabi Legunsen, August Shi, and Milos Gligoric. 2019. A Framework for Checking Regression Test Selection Tools. In *International Conference on Software Engineering*. 430–441.