HAL open science

Rotten Green Tests in Java, Pharo and Python

Vincent Aranega, Julien Delplanque, Matias Martinez, Andrew P Black, Stéphane Ducasse, Christopher Fuhrman, Guillermo Polito, A Etien

To cite this version:

Vincent Aranega, Julien Delplanque, Matias Martinez, Andrew P Black, Stéphane Ducasse, et al.. Rotten Green Tests in Java, Pharo and Python: An Empirical Study. Empirical Software Engineering, 2021. hal-03281836v1

HAL Id: hal-03281836

https://inria.hal.science/hal-03281836v1

Submitted on 8 Jul 2021 (v1), last revised 4 Oct 2021 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Noname manuscript No.
(will be inserted by the editor)

Rotten Green Tests in Java, Pharo and Python

An Empirical Study

Vincent Aranega · Julien Delplanque ·

Matias Martinez ·

Andrew P. Black · Stéphane Ducasse ·

Anne Etien ·

Christopher Fuhrman · Guillermo Polito

the date of receipt and acceptance should be inserted later

V. Aranega, J. Delplanque, A. Etien, G. Polito

Univ. Lille, CNRS, Centrale Lille, Inria, UMR 9189 - CRIStAL, F-59000 Lille, France - E-mail: {firstname}.{lastname}@inria.fr

S. Ducasse

Inria, Univ. Lille, CNRS, Centrale Lille, UMR 9189 - CRIStAL, F-59000 Lille, France - E-mail: stephane.ducasse@inria.fr

M. Martinez

Université Polytechnique Hauts-de-France, LAMIH-UMR CNRS 8201, France - E-mail: matias.martinez@uphf.fr

A. P. Black

Dept of Computer Science, Portland State University, Oregon, USA - E-mail: apblack@pdx.edu

C. Fuhrman

École de technologie supérieure, Montreal, Quebec, Canada - E-mail: christo-

Abstract *Rotten Green Tests* are tests that pass, but not because the assertions they contain are true: a rotten test passes because some or all of its assertions are not actually executed. The presence of a rotten green test is a test smell, and a bad one, because the existence of a test gives us false confidence that the code under test is valid, when in fact that code may not have been tested at all.

This article reports on an empirical evaluation of the tests in a corpus of projects found in the wild. We selected approximately one hundred mature projects written in each of Java, Pharo, and Python. We looked for rotten green tests in each project, taking into account test helper methods, inherited helpers, and trait composition.

Previous work has shown the presence of rotten green tests in Pharo projects; the results reported here show that they are also present in Java and Python projects, and that they fall into similar categories. Furthermore, we found code bugs that were hidden by rotten tests in Pharo and Python. We also discuss two test smells — *missed fail* and *missed skip* — that arise from the misuse of testing frameworks, and which we observed in tests written in all three languages.

Keywords Testing · Rotten Green Tests · Empirical Study · Software Quality

pher.fuhrman@etsmtl.ca

1 Introduction

An important component of software tests is the execution of *assertions* that check that the system under test satisfies some property, for example, that a method returns a specific value, or that certain data are output to a stream. Developers value "green tests," *i.e.,* tests that pass, because such tests increase the developers' confidence that the software is working as expected. The normal assumption is that a green test has executed one or more assertions, and that those assertions were true.

Rotten green tests [13] are tests that are intended by their designer to execute some assertions, but do not actually do so. Such tests are insidious because they pass, giving the *impression* that some useful property is being validated by assertions they contain. In fact, rotten green tests guarantee nothing, and convey a false sense of confidence. The presence of a rotten green test can be seen as a new kind of test smell [15, 30].

Prior work by Delplanque et al. [13] proposed a method to detect rotten green tests in Pharo [5], using a combination of static analysis and dynamic monitoring of method execution at the granularity of individual call sites. Their method determines whether or not a test is rotten, even in presence of helper methods and trait composition [16, 17]. Delplanque et al. evaluated their method on a small sample of 17 Pharo projects. They found that, although their approach does not report false negatives, it can report false positives: tests can be flagged as rotten (*i.e.,* they contain at least one assertion that is not executed) even though they are not defective. False positives can occur

in two situations: in conditional tests, and in so-called flexible tests that are reused in multiple contexts. However, these false-positive situations did suffer from a test smell known as *Conditional Test Logic* [26].

This article extends this prior work in Pharo to two additional languages, chosen using the Tiobe[1] and RedMonk[2] programming language rankings. From the languages ranked in the top three, we selected two object-oriented languages, one statically typed (Java) and the other dynamically typed (Python). It also extends the evaluation of Delplanque et al. [13] to a much larger corpus of Pharo projects. We describe how rotten green tests are detected in the JUnit 4 and unittest/pytest testing frameworks, and report on an evaluation of around a hundred mature projects in each language. Rotten green tests were identified in all three languages, showing that they are not specific to Pharo.

The contributions of this work are as follows:

- a categorization of rotten green tests [13] that applies consistently to Pharo, Java and Python projects;
- a large-scale empirical study of rotten green tests found in mature projects in the Pharo, Java, and Python languages, and a detailed analysis of the results of this study;
- *recursive lazy instrumentation*, a novel way of instrumenting dynamic languages, described in Section 4.3.1; and

[1] https://www.tiobe.com/tiobe-index/

[2] https://redmonk.com/sogrady/2021/03/01/language-rankings-1-21/

– the identification of bugs in the software under test that are hidden by rotten green tests.

Section 2 sets the stage by describing the anatomy of a unit test, distinguishing rotten tests from tests containing no assertions, and showing the problems raised by rotten green tests. Section 3 introduces the automated approach we developed to identify rotten tests, and Section 4 describes its implementation in Pharo, Java, and Python. Section 5 presents some research questions and the methodology used to answer them, while Section 6 presents the results for each of the three languages. Section 7 examines threats to validity. Section 8 discusses some aspects of our approach, and future work. Section 9 presents related work, and Section 10 draws some conclusions.

2 The Problem of Rotten Green Tests

Before defining rotten green tests, we first describe the basics of unit testing. Then, we briefly explain "No-assertion Tests" to help distinguish them from rotten green tests.

2.1 Unit Tests

Unit tests are commonly composed of a test *fixture*, which sets up the system under test, one or more *stimuli*, which exercise the component under test, and one or more *assertions*, which verify some expected property or properties of that component [5, 26].

```
1 class ListOrderedSetTest {
2
3 public void testListAddIndexed() {
4     final ListOrderedSet<E> set = makeObject();
5
6     set.add((E) ZERO);  // Fixture
7     set.add((E) TWO);
8
9     set.add(1, (E) ONE);  //Stimulus
10    assertEquals(3, set.size());  //Assertions
11    assertSame(ZERO, set.get(0));
12    assertSame(ONE, set.get(1));
13    assertSame(TWO, set.get(2));
```

Listing 1 Elements of a unit test.

In Listing 1, taken from the Apache Common Collections project[3], the *fixture* is the code on lines 4–7 that declares and initializes set to contain the constants ZERO, and TWO. Here, the fixture is inline, but it can also be factored out into a setUp method. The *stimulus* is the addition of ONE to set at index 1 on line 9. The first assertion verifies that the size of set is 3; the other 3 assertions verify that the new element (ONE) was correctly inserted, that the first element (ZERO) is still in the same place, and finally that the last element (TWO) was shifted one position after the insertion.

Provided that all of the assertions hold, this test will pass; we say that it is "green". If a false assertion is executed, for example, if the insertion of an element at position p does not shift to the right the element previously located at p, the test will fail; it will be "yellow". If an unexpected error occurs during the running of the test, for example, if makeObject raises an exception, then the test will be "red".

[3] https://github.com/apache/commons-collections/blob/master/src/test/java/org/apache/commons/collections4/set/ListOrderedSetTest.java#L129

```
1 class ListOrderedSetTest {
2
3 public void testListAddIndexed() {
4     final ListOrderedSet<E> set = makeObject(); //Local variable definition
5     set.add((E) ZERO);  // Fixture
6     set.add((E) TWO);
7     set.add(1, (E) ONE);  //Stimulus
```

Listing 2 A no-assertion test does not contain assertions.

2.2 No-assertion Tests

Sometimes, tests have no assertions. This may occur because of the common practice of using a unit-testing framework to execute so-called "smoke tests", also known as "build-verification tests". The purpose of such tests is to check that the feature under test can be run without emitting "blue smoke" (as might be emitted by defective electronics). In software, this means that the smoke test or build-verification test can run to completion without raising an unexpected exception [18]. However, because some of these tests may also contain assertions, for clarity we use the term *no-assertion test*, rather than "smoke test", to refer to a test that contains no assertions. Listing 2 is an example of a no-assertion test.

No-assertion tests can be useful, because if they are green they provide a fast but cursory check that the targeted feature can be considered for further testing. Conversely, if a no-assertion test is red, there is a serious issue that should be addressed rapidly. No-assertion tests are not the focus of this article; nothing that follows should be construed as advocating either for or against the use of no-assertion tests. Nevertheless, we do need to distinguish no-assertion

tests, which *by design* contain no assertions, from rotten green tests, which *by accident* execute no assertions.

2.3 Rotten Green Tests

Consider an empty test — a test method that contains no code at all: no fixture, no stimulus, and no assertion. If it is treated as a passing test, it will increase the number of green tests without providing any value. Empty tests are bad, because they may help to convince a developer that the software is working correctly, when in fact they guarantee nothing. Empty tests do occur — perhaps as the remains of a test-writing session that was never finished. Fortunately, they are easy to spot and eliminate.

A much more insidious problem is caused by a non-empty test with a valid fixture, stimulus, and one or more assertions that — contrary to the intentions and expectations of the programmers — are not executed. We call such tests *rotten green tests*.

Listing 3 is an example (simplified for presentation purposes) of a rotten green test. In this test, a `Logger` instance first logs two entries (lines 3–5). The test then iterates over the entries (lines 6–8) and asserts that for each entry, the logger contains it (line 7). Unfortunately, it turns out that the method `getLogEntries()` does not work as the developer expected: it returns an empty collection, resulting in zero iterations of the `for` loop. This could have happened because there is a bug in the Logger library, or because the developer misunderstood its API. Regardless of the cause, the developer expected the test to

```
1  @Test
2  public void testLoggerContainsLogEntry(){
3      Logger logger = new Logger();
4      logger.log("log1");
5      logger.log("log2");
6      for (LogEntry logEntry : logger.getLogEntries()) {
7              assertTrue(logger.containsLogEntry(logEntry));
8      }
```

Listing 3 A rotten green test. The method getLogEntries() returns an empty collection, resulting in zero iterations, and no executed assertions.

assert the existence of the two log entries that the test added. Not only does this test fail to execute any assertion: the test is *green*.

In the extreme case where all tests are rotten, the coverage of the system under test may be good, but none of the software's features and properties is actually being tested. Moreover, unless developers look quite closely at a test such as the one in Listing 3, they will probably not notice that nothing is actually tested. Consequently, we consider rotten green tests to be harmful: they give the developers the false impression that the code is equipped with meaningful tests.

2.4 A Real-world Example of a Rotten Green Test

We now present a rotten test we discovered in an open-source project created and maintained by Alibaba, one of the biggest technology companies specializing in e-commerce and retail. This project is Sentinel, a flow-control component enabling reliability, resilience and monitoring for microservices. The test case

```
209 @Test
210 public void testListWindowsNewBucket() throws Exception {
211     final int windowLengthInMs = 100, intervalInSec = 1;
212     ...
213     BucketLeapArray leapArray = new BucketLeapArray(sampleCount, intervalInMs);
214     long time = TimeUtil.currentTimeMillis();
215     Set<WindowWrap <MetricBucket> > windowWraps = new HashSet<WindowWrap<MetricBucket> >();
216     windowWraps.add(leapArray.currentWindow(time));
217     windowWraps.add(leapArray.currentWindow(time + windowLengthInMs));
218
219     Thread.sleep(intervalInMs + windowLengthInMs * 3);
220
221     List<WindowWrap <MetricBucket> > list = leapArray.list();
222     for (WindowWrap <MetricBucket> wrap : list) {
223         assertTrue(windowWraps.contains(wrap));
224     }
225     // This won't hit deprecated bucket, so no deprecated buckets will be reset.
226     // But deprecated buckets can be filtered when collecting list.
227     leapArray.currentWindow(TimeUtil.currentTimeMillis()).value().addPass(1);
228
229     assertEquals(1, leapArray.list().size());
230 }
```

Listing 4 A real rotten green test from the Alibaba Sentinel project. A misplaced call to sleep() results in an empty list at line 221, and the assertion at line 223 is never executed.

testListWindowsNewBucket()[4] in the BucketLeapArrayTest class is rotten; the test is shown in Listing 4.

The test *fixture* creates an instance leapArray (of class BucketLeapArray) on line 213. Each element of this array is a "bucket" representing a period of a time, modeled by the class WindowWrap. Instances of WindowWrap contain the starting and ending times of a time interval. The constructor of the array receives the local variable intervalInMs indicating the total time span that the array covers. Once the array is created, the test calls the array's method currentWindow(long time) twice (lines 216 and 217). Each call creates a new time interval (starting at the time passed as argument), and returns the added ob-

[4] https://github.com/alibaba/Sentinel/blob/103fa307e57de1b6660a8a004e9d8f18283b18c9/sentinel-core/src/test/java/com/alibaba/csp/sentinel/slots/statistic/metric/BucketLeapArrayTest.java#L209

ject of type WindowWrap. The test saves these returned objects in a set named windowWraps.

The *stimulus* of the test then calls Thread.sleep, to delay the test's execution for some time, and then creates a list from the array by calling list() (line 221). The *assertions* of the test checks that the elements of this list are indeed the elements added previously. A further stimulus (line 227) adds a new element to the leapArray at the current time (after the delay), and then asserts that this is the only element in leapArray.

At first glance, this looks like a fine test of a BucketLeapArray. Upon running the test, we see it is passing, indicating everything is OK. However, if we look deeper, we see that with the given stimulus, which involves sleeping before obtaining the list() at line 221, the list at line 222 is actually *empty* because too much time has passed. This means that the assertion at line 223 (inside the for) is never executed. Note that the test also does not assert the length of the list before the for statement, although it does so later at line 229. Indeed, at that point the list has its expected size, owing to the manipulation of the leapArray at line 227. Because all *executed* assertions are true, the test passes. However, because the the assertTrue on line 223 is never executed, testListWindowsNewBucket is a rotten green test.

We can hypothesize about how this might have happened. First, leapArray.list() is a time-sensitive call, and likely should have been done *before* waiting with Thread.sleep(), which is a necessary stimulus for line 227. Second, looking at the git history of the file, and at other tests where BucketLeapArray.list() is

called *before* Thread.sleep(), we find that this fault was probably introduced into this test when filtering was added[5].

3 Identifying Rotten Green Test: a Conceptual Perspective

Once we accept that rotten green tests are bad, it is natural to ask how we can detect them. One might think that all that is necessary is to detect tests that make no assertions, but this won't let us distinguish no-assertion tests from rotten green tests. Nor can we assume that any test that contains no assertions directly in its body is a no-assertion test, because a test can make assertions indirectly through the use of helper methods (see Section 3.1).

To clarify the discussion, we will use the following terms:

- An *assertion primitive* is a method provided by the unit-testing framework that performs the actual check. For example, in JUnit 4 there are 56 assertion primitives implemented in the class Assert (*e.g.,* assertEquals(Object expected, Object actual), and assertTrue(boolean condition)). There are two groups of language- and framework-dependent methods that we do not consider to be assertion primitives: a) methods of the testing API that explicitly fail the test, and b) methods that allow the test to be skipped without signaling a failure.
- A *test method* is a method identified as containing a test by the unit-testing framework. For example, this might be any method in a subclass

[5] See commit at https://github.com/alibaba/Sentinel/commit/a65d16083dffd56069c0694d0f5417454d518b22#diff-c85162534c5c25c163e1279cd8f926b7L174

```
1 public void shouldCompileTo(final String template, final Object context, final Hash helpers, final Hash
      partials, final String expected, final String message) throws IOException {
2   Template t = compile(template, helpers, partials);
3   String result = t.apply(configureContext(context));
4   assertEquals("'" + expected + "' should === '" + result + "': " + message, expected, result);
5 }
```

Listing 5 A helper method from the Java project Handlebars for compiling a template and asserting the result. This method is used from other test methods and would not normally be recognized by JUnit as a test method. It contains an assertion on line 4.

of TestCase whose name begins with test, or it might be indicated by an annotation such as @Test.

- A *helper method* is a method that makes an assertion directly (by invoking an assertion primitive) or indirectly (by invoking another helper method), but is not a test method. Developers frequently write application-specific helper methods, as we discuss in Section 3.1.
- A *rotten green test* is a test that passes, contains assertions (either directly, or indirectly through a helper method), but in which at least one of those assertions is not executed.

3.1 Helper methods

Regardless of the language, developers can factor out assertions into helper methods. Listing 5 shows a Java example from the Handlebars project. Helper methods affect our analysis in two ways: we need to know which tests invoke helper methods when we look for rotten tests, and we must take into account the possibility that helper methods, as well as tests, might be rotten.

Like a rotten test, a rotten helper method is one that fails to make an assertion in some or all situations. Because the action of a helper method may

depend on the context (for example, the test fixture, and the arguments to the helper), a helper may behave correctly in one context, but be rotten in another. Thus, in detecting rotten helpers, we need to record the specific test that exposes the issue.

3.2 Classifying Tests

The presence and execution of assertions is the key to classifying tests. There are three situations that a rotten test analysis should identify:

Good test. A test passes and has executed all of its assertions, as well as the assertions in its helper methods.

Rotten green test. A test passes but has not executed at least one of its assertions, or one of the assertions in its helper methods.

No-assertion test. Regardless of its outcome, a test and its helper methods contain no assertions.

Listing 6 illustrates these definitions and shows why detecting rotten green tests can by tricky. Method testABC() is a test method (according to the rules of JUnit4); helper() and secondHelper() are helper methods (because helper() invokes secondHelper(), and secondHelper() invokes an assertion primitive). On line 6, testABC contains a valid assertion assertTrue(**true**), which is always executed. In contrast, the call to helper() on line 5 will not be executed, and consequently the assertion in secondHelper() will not be executed either. Thus, testABC() contains (indirectly) an assertion that is not executed, making it rotten by our

```
class RottenTest() {
  @Test
  public void testABC() {
    //Test method
    if(false) helper();
    assertTrue(true);
  }
  public void helper(){
    //Indirect helper
    secondHelper();
  }
  public void secondHelper(){
    //Direct helper
    assertTrue(b);
  }
}
```

Listing 6 A rotten test that neglects to invoke its helper method, but nevertheless makes a valid assertion.

definition. To detect this, an analysis must first observe that there are two assertion call sites in testABC() (one direct and the other indirect), and must further recognize that one is executed, while the other is not. We say that an analysis with this capability has call-site granularity.

In Listing 7, the assertion at line 6 is executed after the method badhelper() at line 5 is executed. However, the assertion in secondHelper() is not executed, because the condition at line 10 is always false. Therefore, badHelper() is determined to be rotten, as is testDEF().

3.3 Combining Static and Dynamic Analyses

Although the details of the analysis for detecting rotten green tests depend on the language and testing framework, the conceptual outline of the analysis is independent of the language. Any analysis obviously requires a list of the assertion primitives of the testing framework; this is produced manually. Then,

```
class RottenTest() {
  @Test
  public void  testDEF() {
    //Test method
    badHelper();
    assertTrue(true);
  }
  public void badHelper(){
    //Indirect helper
    if(false) secondHelper();
  }
  public void secondHelper(){
    //Direct helper
    assertTrue(b);
  }
}
```

Listing 7 A rotten helper method; *badHelper* contains an assertion (indirectly), but that assertion is never executed.

for each test, our analysis relies on the following five actions, illustrated in Figure 1. Their order depends on the characteristics of the language.

- **Identify the assertion primitives** in the test, and create a list of each primitive and the site from which it is called. Depending on the language, this action can require a static or dynamic analysis.
- **Identify helper methods** invoked from the test, and create a list of each helper and the site from which it is called. Depending on the language, this action can require a static or dynamic analysis.
- **Instrument the call sites** of the assertion primitives and helper methods identified in the previous two actions. The obvious way to do this is to modify the code at each call site to record when it is executed. Because we instrument individual call sites, we do not have false negatives[6].

[6] A false negative would be generated by a call site that the analysis labelled "executed" but that was not actually executed.

- **Execute the test** method, including any inherited test methods, while monitoring:
  1. the outcome of the test (pass, skip, fail, or error), and
  2. whether each *assertion primitive* and *helper* call site has been executed.

  Because we are looking for rotten *green* tests, we consider only passing tests that have not been skipped. A method (test or helper) is considered to be rotten when it contains an unexecuted call site for an assertion primitive or a helper.
- **Generate a Report.** The final report has to take into account the way that methods are reused in the test class hierarchy. A test method may be defined in a superclass and used in one or more subclasses. In general, the test fixture, and thus the meaning of the test, will be different at each place in which a test method is used. So the report must state the class of the rotten test, as well as the method that contains the unexecuted call site. Helper methods are typically reused in many test methods; if a helper method is rotten, all the tests invoking it should be reported as rotten, so that the programmer can understand the context in which the helper fails to make an assertion.

3.4 Categorization of Rotten Tests

Rotten tests can occur only if the execution flow contains branches, which in the languages we consider generally means conditionals and loops. Branching is used for many reasons in tests, including distinguishing varieties of error,

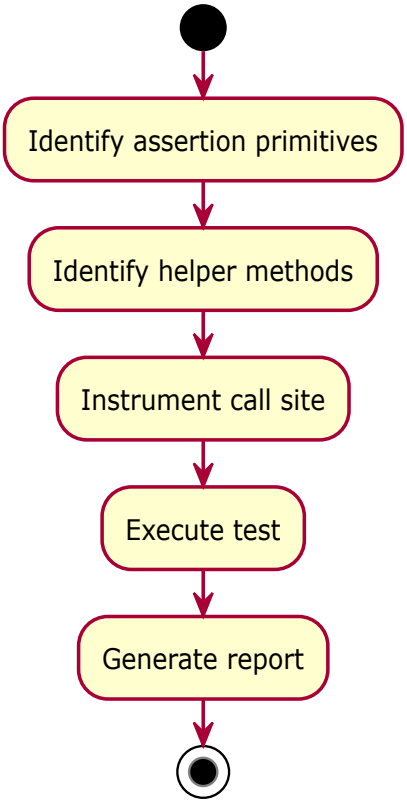

Fig. 1 General approach of rotten-test analysis

and producing better diagnostics. In tests that are designed to be reusable for different configurations of the software, the control flow of the test may be designed so that some branches (and consequently the assertions they contain) cannot be executed. In other words, a test may *by design* contain unexecuted assertions. In such a situation, even though our analysis would classify the test as rotten, it would not have the negative impact of hiding bugs in the test or in the system under test. At the other, more harmful, extreme, loops or branches in tests may be never executed because of faulty logic, resulting in unexpected

empty collections, or conditions that can never be true. We aim to categorize rotten tests to help developers sort through the potentially numerous cases of rotten tests, distinguishing the most egregious cases from the less problematic ones.

After our previous experiments in Pharo [13], we proposed a classification based on four categories: *missed fail*, *missed skip*, *context-dependent assertions*, and *fully rotten tests*. We now review the definitions of these categories.

Missed fail. This category contains tests where the test engineer passes **false** to an assertion primitive to force the test to fail. Such assertion calls are intended to be executed only if something goes wrong, and not if the test passes. This situation is illustrated in Listing 8, line 9, where the assertion primitive assertTrue is called with an argument of **false**. Our approach classifies this test as rotten because the assertion on line 9 is not executed. The intent of the test is that the code in the try block should not raise an exception, but if it does, then the test must fail. To make this test fail, the developer should have used the method fail(String message) available in the API. This can be seen as a misuse of the testing framework. Detecting *missed fail* tests gives the test engineer the opportunity to refactor them to remove these misuses of the API. Once assertTrue(**false**) is changed to fail(...), the test will no longer be classified as rotten. It is easy to identify this anti-pattern in a rotten test, and even to fix it automatically, but this is out the scope of this article.

```
1  public void testHasProtectedConstructor () {
2    try {
3      ConstructorAccess<HasProtectedConstructor> access = ConstructorAccess.get(
         HasProtectedConstructor.class);
4      HasProtectedConstructor newInstance = access.newInstance();
5      assertEquals("cow", newInstance.getMoo());
6    }
7    catch (Throwable t) {
8      System.out.println("Unexpected exception  happened: " + t);
9      assertTrue(false);
10   }
11 }
```

Listing 8 A *missed fail* from project Reflectasm. The intent of the test is that the assertion on line 9 is not executed when the test is green. To reveal this intent, lines 8 and 9 should be replaced with a call to fail("Unexpected exception ..."). It would also be fine, and perhaps clearer, to remove the **try**...**catch** entirely: any test that throws an uncaught exception will automatically fail.

```
1  @Test public void testNormalizedKeyReadWriter() {
2   ...
3   TypeComparator<T> comp1 = getComparator(true);
4   if(!comp1.supportsSerializationWithKeyNormalization()){
5     return;
6   }
7   ...
8   assertTrue(comp1.compareToReference(comp2) == 0);
9  ...
10 }
```

Listing 9 A *missed skip* rotten test in Apache-Flink; if the return on line 5 is executed, then no assertions are made, yet the test passes.

Missed skip. Sometimes test methods contain guards to stop their execution early under certain conditions. Such a pattern is useful when reusing test suites with a variety of fixtures, either to avoid testing a particular configuration, or to avoid running a test that is not appropriate in the current environment. The underlying issue that these tests are trying to work around is an implicit dependency on some other part of the system.

Well-designed tests make such dependencies explicit. The JUnit4 framework provides the Assume class and its primitives, *e.g.,* assumeTrue(boolean b), to improve test design by to allowing test engineers to make dependencies

explicit. But testNormalizedKeyReadWriter in Listing 9 does not use them. Our approach therefore detects this test as rotten.

Once again, this is because of a misuse of the testing framework. To reveal the tester's intention, the guarded return on lines 4 and 5 should be replaced by an assumeTrue(comp1.supportsSerializationWithKeyNormalization()) statement that checks that the comparator under test, comp1, supports the feature to be tested. It is possible to detect this anti-pattern automatically too, by searching for tests containing a return statement and then checking whether that statement was executed. Detecting *missed skip* tests gives developers the opportunity to refactor them so that they take advantage of the testing framework to make their dependencies explicit.

Context-dependent assertion. The *missed skip* category is a special case of tests that contain context-dependent assertions. Whereas a *missed skip* test contains one branch with an early return, other tests contain multiple conditional branches with different assertions in each branch. The boolean expressions in these conditionals are usually checking some property of the environment; in the case of a helper, they may also be checking some property of the helper's parameters. Listing 10 shows an example.

While code like this is not bad *per se*, informing developers of such context-dependent tests can help them improve their tests. *Conditional Test Logic* and *Large Fixture* are two test smells [15, 26, 30] that can have a detrimental effect on test effectiveness and maintenance. Solutions are discussed in Section 8.2.

```
1 @Test public void testCoGroupLambda() {
2  CoGroupFunction<Tuple2<...» f = (i1, i2, o) -> {};
3  TypeInformation<?> ti = TypeExtractor.getCoGroupReturnTypes(f, ...);
4  if (!(ti instanceof MissingTypeInfo)) {
5          assertTrue(ti.isTupleType());
6          assertEquals(2, ti.getArity());
7      ... }
8 }
```

Listing 10 Context-dependent test from Apache flink project (lines 5 and 6).

Fully rotten tests. This category describes tests that do not execute one or more assertions, and do not fall into any of the three previous categories. Fully rotten tests are caused by actual bugs in the system under test, or by errors in the test logic. One logic error that we found multiple times is performing an assertion inside a loop that iterates over an empty data structure. The error in this case is usually located in the code filling the data structure. Listing 4 is an example of a fully rotten test.

To summarize: while our approach definitively identifies tests that contain unexecuted assertions, not all such tests are bugs. The first two categories (*missed fail* and *missed skip*) contain tests in which the test's author failed to take advantage of the primitives provided by the test framework. The *context-dependent* category identifies a test designed to be reused in multiple contexts. These three categories are essentially warnings to developers that indicate opportunities to improve the tests, rather than serious problems. In contrast, the fourth category, *fully rotten tests*, contains dangerous cases; this is where developers should focus their attention.

4 Implementations

We now sketch the separate implementations of the analyses for Pharo, Java, and Python, noting that Java is statically typed whereas Pharo and Python are dynamically typed. Each implementation exhibits different variations of the approach explained in Section 3.3, but the fact that we were able to modify our approach shows that the identification of rotten green tests remains possible across different languages and testing cultures.

4.1 Pharo: Combining Static and Dynamic Analyses

First, we present the implementation for Pharo, which is unchanged from our prior work. For this reason, we keep this description brief; the reader requiring more detail should refer to Delplanque et al. [13].

4.1.1 Analysis

The analysis implemented for Pharo closely follows the description of the approach in Section 3.3 and is composed of five steps from Figure 1. In the following, we describe only what is specific to Pharo.

We first manually identified the assertion primitives in Pharo's *SUnit* framework. This needed to be done just once; the same list of primitives is reused in subsequent analyses.

Identify helper methods. Our analysis identifies helper methods using a static analysis on the test hierarchy. We consider as helpers all methods present

in the test hierarchy that are invoked directly or indirectly by a test method using a **self** invocation. This means that our Pharo implementation does not handle helper methods outside the test class hierarchy. This is an important difference from the Python implementation.

Instrument the call sites. All calls of assertions in the tests and helpers are instrumented using bytecode rewriting to log their execution.

4.2 Java: Combining Static and Dynamic Analyses

In this section we present *RTj* [25], the test identification implementation in Java. It applies to projects that use the testing framework JUnit 4.X.

4.2.1 Analysis

The implementation for Java corresponds to the main steps explained in Section 3.3. Specificities of the Java implementation are detailed below.

Identify the assertion primitives. RTj identifies assertion primitives statically by mining invocations of static methods whose names start with the word assert and where the class of the invocation's target is org.junit.Assert.

Identify helper methods. RTj determines that an invoked method is a helper if it has in its body: (a) an assertion (according to the previous definition), or (b) an invocation of a helper method.

Instrument the call sites. All the code from the application under analysis is instrumented. This instrumentation includes application code, *i.e.,* in the case of a Maven project all the classes inside src/main/java folder, and also the test code (inside src/test/java). Third-party libraries are not instrumented.

Execute the test. RTj executes each test (*i.e.,* those methods with the annotation org.junit.Test). After the execution, for each green test, RTj uses on the instrumentation to determine which lines of code were executed and which were not. It then uses this information to detect unexecuted assertions, and hence to detect rotten green tests.

4.2.2 Computational Cost of RTj

The cost of analyzing a project depends to a large extent on the time taken to execute the (instrumented) test suite. In the Java implementation RTj, test execution is the only step requiring a *dynamic* analysis. The other steps, *e.g.,* identification of code elements, call-site instrumentation, and classification, use *static* analysis, whose computation costs depend on the amount of source code in the tests. In our experiments, the time to execute these static analyses is a small fraction of the time to execute the instrumented tests. The total time taken by RTj to analyze Java projects in our evaluation datasets ranges from 1 to 20 minutes.

4.2.3 Java Example Explained

In section 2.3 we presented a fully rotten test found in the Alibaba Sentinel project. Our tool RTj first identified the assertion written at line 223, which is inside a *for*-loop, and then instrumented it. It then ran the test case, and analyzed the execution of the instrumented code. From this it was able to determine that the assertion was not executed. Finally, because that assertion is inside a loop that does not have an *if-else* as a parent, the test is classified as a *fully rotten*.

4.3 Python: Dynamic Analysis for Rotten Green Identification

Our rotten-test analysis applies to Python projects that use either the unittest[7] framework embedded in the Python default libraries, or the pytest[8] framework. Unlike the Java and Pharo implementations, the analysis for Python is entirely dynamic. First, we will show how the steps described in Section 3.3 are performed by the Python implementation. Then, we briefly explain how the features of this implementation solve problems related to the dynamic nature of Python. Finally, we show two examples of rotten green tests, while exposing the differences between the two Python testing frameworks.

[7] https://docs.python.org/fr/3/library/unittest.html

[8] https://docs.pytest.org/en/latest/index.html

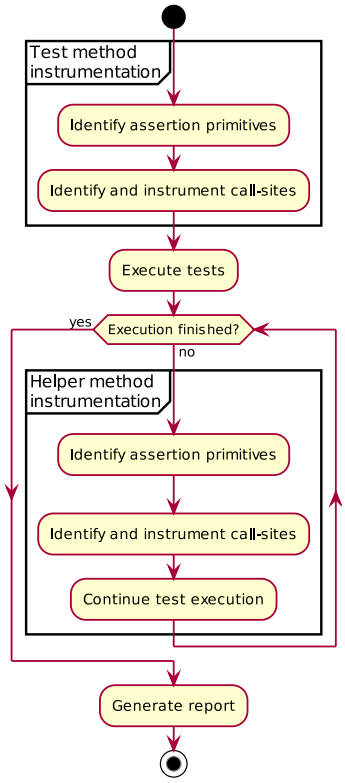

Fig. 2 Python implementation of dynamic analysis for rotten-test identification.

4.3.1 Analysis

As stated in Section 3.3, the rotten-test analysis consists of five actions. However, the order in which these actions are performed is different in the Python implementation, because the majority of helper methods are defined at the module level, outside a test class hierarchy, and because of Python's dynamic nature. For these reasons, identifying helper methods and instrumenting them has to be performed at run time if the analysis is to be precise. Consequently,

the Python analysis starts by instrumenting the test method, and then, at run time, replicates itself into each function or method called by the test.

Instrumenting the code of methods while they are running is an uncommon task that requires specific tooling. In our Python analysis, the test and method instrumentation is performed using the reflectivipy library,[9] a Python implementation of Pharo's Reflectivity library that allows the addition of meta-behavior to any part of an AST at run-time [9].

The Python implementation is illustrated in Figure 2, and consists of two main activities: test method instrumentation, and helper method instrumentation. In contrast to the other implementations, these activities are performed at different times. First the test method is instrumented with self-propagating instrumentation assertions. The test method is then executed. During execution, the instrumented code calls code that recursively instruments each method and function called from the test, until the test method execution terminates.

These two main activities are composed of two sub-activities: *Identify assertion primitives* and *Identify and instrument call sites*. Sub-activities with the same name are basically the same, but occur at different moments during the execution of the test.

Identify the assertion primitives. Assertion calls are easily detected on each of the target testing frameworks. In the case of the unittest framework, assertions are methods inherited from the TestCase class. They are limited to

[9] https://github.com/StevenCostiou/reflectivipy/

a set of known methods, such as assertTrue, and assertFalse.[10] An assertion call is characterized by two conditions: the method name must be in the set of assertion method names, and the object on which the method is called must be an instance of TestCase. Detecting assertion calls is done just before executing the assertion primitive, when the class of the object on which the method will be called is known. In the case of the pytest framework, assertions are made using Python's assert keyword, making detection straightforward: the AST for the test method is scanned at run time, and each assert node is instrumented.

Identify and instrument the call sites. Before test execution, the AST of the test under analysis is scanned and each method and function call is instrumented to add a meta-behavior prior to the execution of the call. This meta-behavior is responsible for two tasks:

- recursively "spreading" the instrumentation over the called method or function, by instrumenting its body at run time;
- identifying and instrumenting each assertion primitive to check if it has been executed.

In short, the instrumentation propagates itself over the code at run time, when it knows the exact class of the object on which the methods will be executed. We call this way of instrumenting code and propagating the instrumentation dynamically *recursive lazy instrumentation*.

[10] A list of the assertion methods in unittest can be found at https://docs.python.org/3/library/unittest.html

Although dynamic instrumentation circumvents the limitations of static analysis, it introduces the need to limit the propagation of the instrumentation. Indeed, if recursive lazy instrumentation were to propagate through *every* method call, at some point it would reach either Python core methods, or methods from third-party libraries, both of which are out of the scope of our analysis. To avoid this situation, we constrained the instrumentation to methods from modules that are located in the tests of the subject project, or in the tests of third-party libraries.

Execute the tests. The analysis is implemented as a pytest plug-in that gathers all tests discovered by the pytest and applies recursive lazy instrumentation to them. With the recursive lazy instrumentation initialized, the tests are launched using the default pytest runner; pytest can execute tests written for unittest as well as pytest.

Our method of detecting helper methods has one limitation: it cannot identify helper methods that raise Python exceptions rather than making assertions. We found that, in some projects, there were helper methods that did not use any kind of assertion, but instead conditionally raised an exception. In these projects, raising an exception is a pattern that emulates an assertion mechanism. Those methods cannot be classified as helper methods or assertion primitives by our approach, as there is no way to be sure of how the exception will be handled in every test.

```
1 def test_moved_attribute(self):
2     attr = six.MovedAttribute("spam", "foo", "bar")
3     if six.PY3:
4         assert attr.mod == "bar"
5     else:
6         assert attr.mod == "foo"
7     assert attr.attr == "spam"
8     attr = six.MovedAttribute("spam", "foo", "bar", "lemma")
9     assert attr.attr == "lemma"
10    attr = six.MovedAttribute("spam", "foo", "bar", "lemma", "theorm")
11    if six.PY3:
12        assert attr.attr == "theorm"
13    else:
14        assert attr.attr == "lemma"
```

Listing 11 A rotten green test with the pytest framework.

4.3.2 Python Examples Explained

In this section we examine an example from each of the Python testing frameworks that were part of our study. Listing 11 shows a context-dependent rotten test written with the pytest framework. This test was extracted from the six library, which provides a way of writing code compatible with Python 2 and Python 3. The analysis (which was run in a Python 3 environment) reports that line 6 and line 14 are not executed. Those assertions are part of an if statement and are executed only in the case of Python 2. This is an example of a *context-dependent* rotten green test. Note that it is also an example of the *Conditional Test Logic* smell, with the cause being a so-called *Flexible Test* [26].

Listing 12 shows a test written with the unittest framework from the impacket project, a collection of classes for working with network protocols. The method is in a dedicated class that inherits from TestCase (line 1). The analysis reports that the assertions on lines 10 and 14 are not executed. We can see that there

```
1 class TestDot11Decoder(unittest.TestCase):
2   def setUp(self):
3     self.WEPKey=None #Unknown
4     # other inits
5
6   def test_04_Dot11WEPData(self):
7     if not self.WEPKey:
8       return
9
10    self.assertEqual(str(self.in3.__class__), "impacket.dot11.Dot11WEPData")
11
12    # Test if wep data "get_packet" is correct
13    wepdata=b"\x6e\xdf\x93\x36\x39\x5a\x39\x66\x6b\x96\xd1\x7a\xe1\xae\xb6\x11\x22\xfd\xf0\xd4\x0d\x6a\
        xb8\xb1\xe6\x2e\x1f\x25\x7d\x64\x1a\x07\xd5\x86\xd2\x19\x34\xb5\xf7\x8a\x62\x33\x59\x6e\x89\x01\
        x73\x50\x12\xbb\xde\x17'
14    self.assertEqual(self.in3.get_packet(),wepdata)
```

Listing 12 A rotten green test with the unittest framework.

is an explicit return on line 8 that is executed when the WEP key is not set. This rotten test is classified as a *missed skip*, meaning that a skip instruction from the testing framework should have been used instead of the conditional return. Deeper examination of the code shows that in the method used to set up this test, the WEP key is explicitly set to None, implying that this test will *never* be executed.

4.4 The Tool Development Process

To ensure that the three implementations label the tests in the appropriate way, we developed and then fine-tuned them by iterating on a small set of test projects. On those projects, we manually checked to see that all rotten tests were labeled with the correct category. We also checked that the tests *not* reported as "rotten" really had their assertions executed; we did this by manually inserting a breakpoint in each assertion marked as executed. Once each implementation was fine-tuned, we executed it on all of the language-

appropriate test projects, and manually checked the labelling. Across all the tests, we did not find any that was incorrectly labeled as fully rotten (*i.e.,* we did not find any false positives).

5 Evaluation Setup

In this section we state the research questions that our empirical analysis was intended to answer. We also explain the criteria and methodology behind our selection of projects for evaluation.

5.1 Research Questions

In previous work [13], we identified the existence of rotten green tests in Pharo projects, and categorized them by cause and by potential for danger. Our intuition made us suspect that rotten tests were not specific to Pharo, and our definition of rotten green tests was intentionally language-independent. However, intuition is no substitute for data: we wanted to find out whether or not rotten green tests actually exist in projects written in other languages, and whether they are more or less common than in Pharo projects. After all, rotten tests might be peculiar to Pharo, perhaps as a consequence of some mis-feature of Pharo's unit testing framework.

One of the goals of the present evaluation, then, is to see whether or not our earlier findings hold for other languages. For this purpose, we choose Java and

Python, two languages considered to be popular by programming language rankings such as PYPL.[11]

If rotten green tests do exist in other languages, we then want to find out if our previous categorization [13] is also valid for these languages. In other words, are we able to identify *missed fail*, *missed skip*, *context-dependent assertions*, and *fully rotten tests* for each studied language? Do these languages present additional or alternative categories that were not present in Pharo?

If the categories are the same, are the distributions of rotten tests over the categories also the same? In Pharo we found lots of *context-dependent* and *missed fail* tests, fewer *missed skip* tests, and even fewer *fully rotten* tests. By analyzing the *fully rotten* tests in Pharo, we determined that most of these tests were badly written, and that some hid bugs in the code under test. Do similar situations exist in the other languages?

We summarize these concerns with the following research questions.

RQ1: Is the same categorization of rotten green tests applicable to Java, Pharo and Python?

RQ2: In Java, Pharo and Python, to what extent do rotten green tests hide bugs in the program under test, rather than bugs in the tests themselves?

[11] PYPL (PopularitY of Programming Language Index) https://pypl.github.io/PYPL.html was created by analyzing how often language tutorials are sought on Google. By this metric, Python and Java were the two top programming languages in March 2021.

5.2 Project Selection Methodology

To answer these research questions, we selected around 100 projects per language (Java and Python) from among the most popular projects on GitHub. Since we wanted to find projects representative of each language and its community, we did not want to base our selection only on GitHub's star ratings. Consequently, we added conditions on project activity in terms of recent commits, forks, pull requests or closed issues. First, only projects with a *push* after 2019/1/1 were considered. Second, to account for a project's representativeness, we computed a score for each project using the expression

$$(mergePullRequestsCount + closedIssuesCount)/10 + forkCount + starsCount$$

We sorted the projects by score and selected the top hundred projects.

In addition, because the of implementation of our analysis differs across the languages, we added some language-specific requirements, described below.

Extra criteria for Pharo projects:

- Pharo had to be the main programming language.
- Pharo is a small community in comparison to Java and Python. To obtain a corpus of similar size (around 100 projects) we relaxed the date condition to push dates after 2017/1/1.
- The version of Pharo had to at least 5.0 (released in 2015), the oldest version of Pharo for which our implementation works.
- The project had to be configured with continuous integration.

- It had to be possible to run tests locally, and all tests had to pass.

We extracted 113 projects meeting these criteria.

Extra criteria for Java projects:

- Java had to be the main programming language.
- JUnit 4.x had to be the testing framework.
- Maven had to be used for dependency management, because RTj is able to compile Maven projects automatically.

We successfully executed RTj on 67 of the selected projects. Two major issues prevented us from analyzing the remaining projects. For some projects, we could not automatically build a version using Maven — for example, the project required the creation of environment variables dedicated to the project. Other projects required Java 11 or a more recent version; RTj does not support the analysis of the Java bytecodes from version 11 onwards.

Extra criteria for Python project selection:

- Python had to be the main programming language.
- The testing framework had to be pytest or unittest.
- The project had to have a setup.py file.

We selected the first 100 projects matching these criteria. Among these projects, some were "template" projects, *i.e.,* projects used as a demonstration of a well-initialized project; we excluded them. We finally identified 96 projects on which we ran the analysis.

Next, we applied to each project the tools described in Section 4, and gathered the resulting data[12]. These data helped us answer research question *RQ1*. Manual investigation of the identified rotten tests and helpers was needed to answer *RQ2*.

6 Results

This section separates the results into two aspects. First we look at the research questions (Section 6.1), and second we report the results of the manual assessment of the identified rotten green tests. We present both aspects for each language (Sections 6.2, 6.3, 6.4).

6.1 Answers to the research questions

6.1.1 RQ1*: Presence of rotten tests and their categorization*

Table 1 shows, for each language, the number of rotten tests and the number of affected projects in each category. The last two columns present the number of projects with rotten tests and the number of projects analyzed.

The table shows that rotten green tests are present in projects in all three languages: in Python and Java projects as well as in Pharo projects. Indeed, the data show that rotten green tests are much more common in Python, and somewhat more common in Java, than they are in Pharo. After analyzing a

[12] Raw data are available at https://github.com/rmod-team/2020-rotten-green-tests-experiment-data

Language	missed fail		missed skip		context-dep.		fully rotten		Projects
	Tests	Proj	Tests	Proj	Tests	Proj	Tests	Proj	Rotten/Analyzed
Pharo	180	6	57	8	321	7	107	6	17/113 = 15%
Java	13	5	51	7	254	18	100	16	26/67 = 38%
Python	212	12	71	17	831	30	50	15	41/96 = 43%
								Total	85/276 = 31%

Table 1 Number of rotten tests (Tests) in each category, and number of projects (Proj) containing such tests. The last two columns show how many projects contained rotten tests, and how many projects were analyzed.

total of 276 projects, we found 85 projects, or around 31 per cent, contained rotten green tests. In more detail:

Pharo. We found 665 rotten green tests in 17 projects. Thus, 15% of Pharo projects (17 out of 113) in our dataset have at least one rotten green test. A significant proportion of rotten green tests are from the context-dependent category (321 tests from 7 projects) and the missed fail category (180 tests from 6 projects). Just 6 projects (5% of the analyzed projects) contain fully rotten tests.

Java. We found 418 rotten green tests in 26 projects. This means that the 38% of the projects analyzed (26 out of 67) have at least one rotten green test. The majority of the rotten tests found are from two categories: 254 context-dependent tests from 18 projects, and 100 fully rotten tests from 16 projects (a total of 23 distinct projects). This means that around one

out of three projects (23/67) presents either context-dependent or fully rotten green tests.

Python. We found 1164 rotten green tests in 41 projects. Thus, nearly 43% of the projects analyzed (41 out of 96) contained at least one rotten green test. The majority of these rotten tests were from two categories: 212 missed fail tests from 12 projects, and 831 context-dependent tests from 30 projects. Nearly 16% of the projects we analyzed contain fully rotten tests (15 out of 96).

As illustrated by Table 1, each of the four categories of rotten green tests exists in the three studied languages, answering *RQ1*. Because the fully rotten category is defined in opposition to the other three, any rotten test that does not belong to categories *missed fail*, *context-dependent* or *missed skip* is considered fully rotten by the tool. For all languages, the most common category is "context-dependent".

The usefulness of this categorization becomes apparent when we consider what a test engineer should do when presented with the data from our analysis tools. They should examine the fully rotten tests, since these are likely to camouflage bugs. Fortunately, this is a small percentage of the total — 16% for Pharo, 24% for Java, and 4% for Python. The remaining categories of rotten tests are less crucial. Rather than being bugs, they can be considered to be test smells that indicate a misuse of the testing framework, and thus an invitation for refactoring.

Figure 3 presents, for each project containing a fully rotten test, the ratio of fully rotten tests to total tests. We observe that in all the Pharo projects, there is one that contains a high proportion (0.28) of fully rotten green tests. This is the *Collections* project, with 17 fully rotten green tests out of 59 total tests. We can see that in the three languages, the ratio of fully rotten green tests is low, indicating that fully rotten green tests are rare. The number of projects actually containing fully rotten tests is also low: from 6 to 16. Consequently, it is difficult to draw more conclusions from these data.

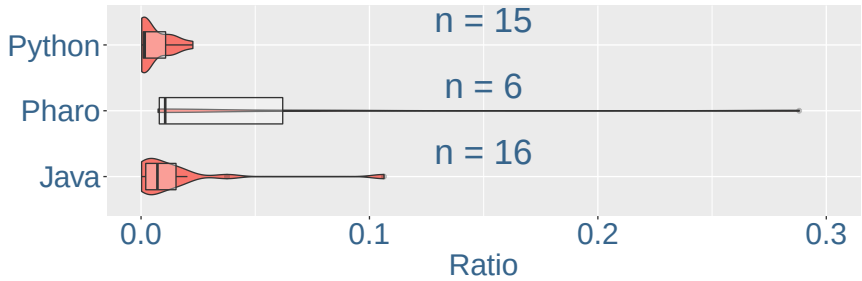

Fig. 3 Violin graph presenting ratios of fully rotten green tests to total tests among projects that contain at least one fully rotten green test.

6.1.2 RQ 2: Do rotten green tests hide bugs in the program, or only in the tests?

Discovering a rotten green test raises a doubt, because the test is either testing nothing, or testing only part of the code. This could mean that the code under test contains errors that are not exposed by the (rotten) tests. Finding actual bugs in these projects is challenging for researchers outside the project, as it

requires knowledge of the project, the desired behavior of the code, the tests, and the features addressed by the tests. Consequently, discovering an actual bug in the software under test that is concealed by a rotten green test remains a demanding manual task.

For each fully rotten test case, we manually fixed the test so that the assertions were executed, and then checked to see if the test remained green. If it did, the properties checked by the test did actually hold, meaning that program under test behaved as intended by this test case. (This assumes that the test case actually asserts the intended properties, which is something we cannot know.) If the test turned red, something was wrong in the program under test: it violated one of the test's assertion. The modifications that were needed to force a test to execute previously unexecuted assertions were unique for each test; we did not find a way to generalize them.

In this section, we discuss some examples of rotten green tests that hid bugs in the program under test. Our goal is not to identify or quantify all bugs hidden by rotten green tests, but merely to establish that such bugs exist.

As mentioned by Delplanque et al. [13], four fully rotten green test identified in the Pharo Compiler project hid a bug in a rarely used runtime code transformation. The rotten tests were intended to check the behavior of the dynamic bytecode rewriting of Boolean expressions. In most Smalltalks, built-in Boolean objects are treated specially. Consequently, it is impossible to make a (user-defined) object act like a Boolean, even though it implements the

```
1 MustBeBooleanTests » testAnd (original)
2    | myBooleanObject |
3    myBooleanObject := MyBooleanObject new.
4    self deny: (myBooleanObject and: [true])
5
6 MustBeBooleanTests » testAnd (rewritten)
7    | myBooleanObject |
8    myBooleanObject := MyBooleanObject new.
9    ^ (myBooleanObject) and: [ 1 halt ]
```

Listing 13 A rotten test in Compiler in its original form, and as it is decompiled after bytecode rewriting. MyBooleanObject new creates a new user-defined object that behaves like **false**, so the **self** deny should hold. However, a bug in the rewriting process means that the deny assertion is never executed..

Boolean interface. This is because common boolean methods such as ifTrue: and and: are compiled to optimized bytecodes that work for built-in Booleans but which fail (by raising an exception) when evaluated on non-Booleans. However, Pharo catches this exception and rewrites the bytecodes dynamically; it uses a deoptimization that actually sends the message (*e.g.,* ifTrue:) to the non-Boolean receiver.

A bug in the bytecode-deoptimization process introduced an early return into these four tests. Listing 13 shows the source code of one of these methods, both in its original form, and as it was decompiled from the dynamically deoptimized bytecode. We see that the bytecode-deoptimization process introduces a *return* (^) in place of the assertion on line 9. This leads to the four tests turning green, because no assertion is executed. This example shows that it can be extremely difficult to see that an assertion is not executed. It was the only case that Delplanque identified of a bug being hidden by a rotten green test; we did not find any new Pharo bugs in the current study.

```
1 def test_double_extends(self, env):
2 """Ensures that a template with more than 1 {% extends ... %} usage
3 raises a ``TemplateError``.
4 """
5  try:
6     tmpl = env.get_template('doublee')
7  except Exception as e:
8     assert isinstance(e, TemplateError)
```

Listing 14 A fully rotten test in the *jinja2* project, caused by bad exception handling.

In Python, we did find bugs hidden by a rotten test. Listing 14 shows a test written for the *jinja2* project[13], a widely used template engine written in Python[14]. This test ensures that if a template contains more than one extends, a dedicated TemplateError is raised. The rotten-test analysis showed that the assert at line 8 was never executed, indicating that the exception was never raised. To correctly test this case, the test should have been written either with an else branch on the try ... except (the else branch would be executed when no exception is raised), or using the pytest.raises context manager, which expects a dedicated exception to be raised. Manually modifying the code of the test by introducing an else branch or the pytest.raises context manager was enough to turn the test red, revealing the bug.

Note that this test was actually identified as problematic and fixed by the developers a few weeks later,[15] in October 2019.

We also inspected the rotten green tests found in Java, but were not able to find any bugs hidden by a rotten test. As previously stated, finding a bug that could be hidden by a rotten green test is challenging, especially for developers

[13] https://github.com/pallets/jinja

[14] Repository accessed September 2019, commit 91a404073acac40a7945bf7d584e8b30bc7a08cb

[15] with commit 9ca80538d9e9418ae658772516f9b7dfb1e02ccd

```
1 public static void assertCode(String expectedCode, Object o) {
2       if (expectedCode == null) {
3           assertNull(o);
4       } else {
5           CodeAssertable codeAssertable = convertToCodeAssertable(o);
6           assertCode(expectedCode, codeAssertable);
7       }
8   }
```

Listing 15 Optaplanner project's assertCode helper producing Rotten Green Test.

unfamiliar with the code under test. Thus, our inability to find such a bug does not mean that no such bug exists.

6.2 Further analysis of Pharo projects

We did not perform a deeper analysis of the rotten green tests found in Pharo projects, because such an analysis was performed as part of the prior work [13].

6.3 Further analysis of Java projects

Optaplanner is the Java project with the largest number (110) of rotten test cases. Most of the rotten tests (103 cases) are from the context-dependent category. We found that most of those cases are caused by a particular rotten helper method (assertCode) that is invoked by several test cases. The assertCode helper, shown in Listing 15, has a rotten assertion inside an if ... else. When the value from argument expectedCode is null, the helper executes the assertion on line 3; when the argument is not null, it executes the helper method assertCode on line 6. It is never the case that both line 3 and line 6 are executed.

Other rotten tests of this category found in Optaplanner have the same characteristic: they contain an if ... else with assertions in both branches. In

```
@Test
public void testMapLambda() {
 MapFunction<Tuple2<Tuple1<Integer>, Boolean>, Tuple2<Tuple1<Integer>, String» f = (i) –> null;

 TypeInformation<?> ti = TypeExtractor.getMapReturnTypes(f, NESTED_TUPLE_BOOLEAN_\TYPE, null,
      true);

 if (not(ti instanceof MissingTypeInfo)) {
  assertTrue(ti.isTupleType());
  assertEquals(2, ti.getArity());
  assertTrue(((TupleTypeInfo<?>) ti)....);
  assertEquals(((TupleTypeInfo<?>) ti).getTypeAt(1)...);
 }
}
```

Listing 16 Context-Dependent rotten test found in Java.

this case, the test cases that invoke the helper control which branch is executed by passing a boolean value.

The project with the second-largest number of rotten tests is Flink-core; here, too, the majority of the rotten tests are context-dependent. One example is presented in Listing 16. All the assertions from this test are written inside the then branch of an if. The documentation of the class that includes this test specifies why those assertions are sometimes not executed: *"Many tests only work if the compiler supports lambdas properly otherwise a MissingTypeInfo is returned."* Thus, the execution of the assertion depends on the execution context — in this case the version of the compiler. However, the test passes even when the context is not adequate for executing it.

Other rotten tests from the Flink project are caused by the class hierarchy of test cases. Test cases implement the behavioral design pattern *Template Method*. Flink contains several abstract classes with test cases. For instance, the abstract class ComparatorTestBase aims at testing comparators for various implementations of number (Float, Int, BigDecimal, *etc.*) It contains 11 test cases

```
1 @Test
2  public void testNormalizedKeysGreatSmallAscDescHalfLength() {
3
4  TypeComparator<T> comparator = getComparator(true);
5  if (not(comparator.supportsNormalizedKey())) {
6   return;
7  }
8  testNormalizedKeysGreatSmall(true, comparator, true);
9  testNormalizedKeysGreatSmall(false, comparator, true);
10 }
11
12
```

Listing 17 Skip rotten test found in Java

and one abstract method (getComparator) that returns a comparator. Thus, subclasses of ComparatorTestBase need to override that abstract method to return an appropriate comparator. For instance, the test class *IntComparatorTest* returns an *IntComparator*.

We found some rotten test cases in ComparatorTestBase. Depending on the comparator under test, some tests execute no assertions. For example, Listing 17 has a return statement on line 6 that is executed when the comparator under test does not support normalized keys. Such test cases are *missed skip* rotten green tests; they pass, yet no assertion is ever executed. We discuss such cases later as a potential misuse of the testing framework, because such a test should be skipped, *e.g.,* by using Assume.assumeTrue(**false**) in JUnit 4.

The project with the third-largest number of rotten tests is Streamex, a library for enhancing Java 8 Streams.[16] Similar to the previously described projects, most of the rotten tests are context-dependent rotten tests. One example of these is test testJdk9Basics(), presented in Listing 18. The test contains

[16] https://github.com/amaembo/streamex/blob/1190608bda70885f55ec791ebc0e76f89006db6a/src/test/java/one/util/streamex/InternalsTest.java\#L59

```
@Test public void testJdk9Basics() {
    MethodHandle[][] jdk9Methods = Java9Specific.initJdk9Methods();
    if (Stream.of(Stream.class.getMethods()).anyMatch(m -> m.getName().equals("takeWhile")))
        assertNotNull(jdk9Methods);
    else
        assertNull(jdk9Methods);
  }
```

Listing 18 Example of context-dependent assertion in Java

an if with assertions in the two branches, one branch for Java 9 (or newer) code, and the other for older versions. The test executes only one branch, depending on the JDK used to run it; the if condition looks for the takeWhile method on streams, which was introduced in Java 9. Consequently, which assertion is executed depends on the *execution context*.

6.4 Further analysis of Python projects

Here we present some rotten tests that are representative of common mistakes found in many of the analyzed projects.

The Python project that contains the most rotten tests (536) is *aws-cli*. Among them, the commonest category is context-dependent tests, many of which arise because this project is compatible with both Python 2 and Python 3. In this project, we spotted only 3 fully rotten tests; they occur for two reasons.

The first reason is bad indentation of a test in the test suite. In Python, blocks of code are delimited by indentation. Python allows functions to be nested inside other functions, so whether a function definition exists at the module level, or nested inside another function, depends on its indentation.

```
1 def test_show_one_call_present(self):
2     self.parsed_responses = [
3         {"Regions": [{"Endpoint": "ec2.ap-south-1.amazonaws.com",
4                       "RegionName": "ap-south-1" },]}]
5     _, _, rc = self.run_cmd('ec2 describe-regions', expected_rc=0)
6     self.history_recorder.record('CLI_RC', rc, 'CLI')
7     self.run_cmd('history list', expected_rc=0)
8     self.assertIn(b'ec2 describe-regions', self.binary_stdout.getvalue())
9     def test_multiple_calls_present(self):
10        self.parsed_responses = [
11            {"Regions": [{"Endpoint": "ec2.ap-south-1.amazonaws.com",
12                          "RegionName": "ap-south-1"},]},
13            {"UserId": "foo",
14             "Account": "bar",
15             "Arn": "arn:aws:iam::1234567:user/baz"}]
16        _, _, rc = self.run_cmd('ec2 describe-instances', expected_rc=0)
17        self.history_recorder.record('CLI_RC', rc, 'CLI')
18        _, _, rc = self.run_cmd('sts get-caller-identity', expected_rc=0)
19        self.history_recorder.record('CLI_RC', rc, 'CLI')
20        self.run_cmd('history list', expected_rc=0)
21
22        self.assertIn(b'ec2 describe-regions', self.binary_stdout.getvalue())
23        self.assertIn(b'sts get-caller-identity', self.binary_stdout.getvalue())
```

Listing 19 A fully rotten test in the *aws-cli* project caused by a badly indented block.

In the test named test_show_one_call_present and presented in Listing 19, we notice that from line 9 to 23, a new function test_multiple_calls_present is defined. This function is intended to be a top-level test method, but because of the indentation, it is actually nested inside the enclosing test_show_one_call_present test. The naming of the method, starting with test_, and the fact that the first parameter is named **self**, are both strong indicators that this nested function was intended to be a test method. Moreover, this nested function is never used in the code of the enclosing method — which is the only scope in which it is accessible. Consequently, we have no doubt that this function was intended to be a top-level test method. However, because it is defined as a nested function, it will not be discovered by the testing framework, and so neither it nor any of its assertions will be executed.

```
1 def test_start_session_when_check_call_fails(self, mock_check_call):
2     # [...] here other part of the test
3     with self.assertRaises(ValueError):
4         self.caller.invoke('ssm', 'StartSession', start_session_params, mock.Mock())
5
6         self.client.start_session.assert_called_with(**start_session_params)
7         self.client.terminate_session.assert_called_with(**terminate_session_params)
8
9         # [...] here final part of the test
```

Listing 20 A fully rotten test in the *aws-cli* project because of a misuse of the testing framework.

The two other fully rotten tests from this project are related to a misuse of the testing framework when dealing with exceptions. Listing 20 presents part of the fully rotten test named test_start_session_when_check_call_fails. In this test, we can see at line 3 that a context manager is used with a special assertion call from the testing framework: **self**.assertRaises(...). This call, used in conjunction with a context manager, will evaluate a block of code and expect a particular exception, in this case a ValueError exception. The logic of this test is to ensure that an exception is raised, and then to check some special values. In this case, line 4 raises a ValueError exception, as expected. As a consequence, the assert_called_with() assertions on lines 6 and 7 are never executed. Due to the semantics of the **self**.assertRaises(...) instruction, we are sure that these two assertions were meant to be executed. If the execution of line 4 in the with context manager does not raise a ValueError exception, **self**.assertRaises(...) will fail. Consequently, the two assertions cannot be used as "failing" assertions, making the test fail if no exception is raised by line 4.

This example demonstrates a problematic pattern in the unittest framework: the use of **self**.assertRaises(...) as a context manager. The same pattern can

be found in the pytest framework, and is identified as problematic in pytest's documentation[17]:

> "When using pytest.raises as a context manager, it's worthwhile to note that normal context manager rules apply and that the exception raised must be the final line in the scope of the context manager. Lines of code after that, within the scope of the context manager will not be executed."

Among all the tested Python projects, the one with the most fully rotten tests is the mlextend project. This project contains 14 fully rotten tests, all of which are related to the use of an exception-handling context manager as described above.

Although most of the errors in the studied tests derive from misuse of the testing framework, there are some that are related to either badly initialized variables (as shown in Section 4.3) or badly indented blocks of code. Among the 96 projects, only a few projects contain fully rotten tests; most of the problematic tests we identified were no-assertion tests and context-dependent tests.

The context-dependent tests were, almost always, found in projects that supported both Python 2 and Python 3. As some features of Python 3 are not compatible with Python 2, a common idiom is to place the code that requires Python 2 in an if statement.

[17] https://docs.pytest.org/en/latest/reference/reference.html#pytest.raises

Fully rotten tests are quite rare in the Python projects we studied; most of them are caused by the context-manager exception pattern. The other fully rotten tests are caused by iterating over empty collections, but they represent only a small fraction of the fully rotten tests.

Fully rotten green tests can hide bugs, but such tests are rare when compared to the total number of tests in each project, as we saw in Figure 3. However, we can conclude that the larger the proportion of fully rotten green tests in a project, the greater the probability of hidden bugs.

7 Threats to Validity

This section discusses the validity of our case study using the validation scheme defined by Runeson and Höst [31]. We discuss construct, internal, and external validity, in that order.

7.1 Construct Validity

Construct validity indicates whether the studied measures really represent what is investigated according to the research questions.

Different implementations. As explained in Section 4, we developed three implementations of our rotten green test analysis, one per language. Each implementation is an adaptation of the generic approach outlined in Section 3.3. It was not possible to use exactly the same approach because of the specifics of each language (statically versus dynamically typed) or the practices of the

community. For example, in Pharo, developers frequently define test helper methods in the test class or in one of its superclasses. In contrast, Python developers tend to define helpers in a separate class hierarchy, making their static identification impossible. Nevertheless, despite the variations in implementation, the definition of rotten tests is the same for the three languages. To mitigate the threat to validity from differences in the implementations, the developers of the analyses — each of whom is an expert in their respective language and who have belonged to the same research group for several years — collaborated closely on the development of their implementations.

Project selection. Our objective was to choose a *representative* sample of projects for each community. As it is often the case when some open-source projects on GitHub are chosen, we put some criteria on the presence of tests, on popularity (number of stars), and on activity. Furthermore, to ease the installation of the chosen projects, we added constraints on the presence of configuration files (*setup.py* for Python and *pom.xml* for Java). Because the number of projects per community is different, we did not select exactly the same number of projects in each language, but we consider that with more than 60 projects in each language we could draw some valid conclusions.

Even if had we tried to use the same criteria to select the projects regardless of the language, it would not have been possible, because the popularity of the languages is not the same. For example, the limit on the number of stars exists only for the Python and Java projects (> 1000). Adding a constraint on stars would not have allowed us to find 100 relevant open-source projects in Pharo.

Similarly, we relaxed the constraint on dates: the date of last push is 2019/1/1 for Python and Java, and 2017/1/1 for Pharo. Although these criteria are thus different for each language, they provided a selection of between 60 and 100 acceptable projects per language. The research questions are independent of the selection criteria.

Configuration problems. When loading some projects, we encountered configuration problems, some of which we were not able to solve. For example, in the Java experiment we faced the following problems when building Maven projects: (a) Unresolved dependencies (the Maven build could not find some dependencies); (b) The build process for a project failed (*i.e.,* the command `mvn compile` failed); (c) The project would not run on the Java Virtual Machine used in our experiment; (d) Environment variables required by a particular project were missing.

Manual Analysis of Discovered Rotten Green Tests Our automated analysis tools might possibly produce *false positives*, i.e., they might label a test as rotten when it not. To check for the presence of false positives, we manually analyzed each rotten case detected by our tools. In addition to the name of the test, our tools also indicate the location l (*i.e.,* file and line number) of the assertion (or helper) that was not executed and, consequently, the one that caused the test to be classified as rotten. Using that information, we opened the project that contained the rotten case in an IDE (Pharo, Eclipse, or PyCharm). Then we put a breakpoint on location l, and executed the test

marked as rotten in mode debug. If the execution breaks at location l, it means the rotten test does in fact execute that statement, and our tools have reported a false positive. Conversely, if execution does *not* stop on that break point, the statement is not executed by the test: we have found a true positive.

This manual analysis did not find any false positive in our corpus of Pharo, Java and Python projects, but does not exclude the possibility that false positive might be found in other projects.

Python Analysis Limitations. As we stated in Section 4.3, our automated analysis may classify some tests as no-assertion tests because of the way those tests are built. In the analyzed repository, we found some tests that raise an exception in a helper method to emulate a failing assertion. During test execution, the test framework catches this exception and marks the test as failing. The automatic categorization classifies these tests as no-assertion tests, since they do indeed contain no assertions. Our analysis fails to recognize them automatically; doing so would require that our automated tools understand the intention behind raising the exception. When we analyzed the results of our experiments, we examined all the no-assertion tests manually, and classified them as rotten only when that was appropriate.

7.2 Internal Validity

Internal validity indicates whether variables other than those studied may have affected the result. Since we do not examine causal relations, internal validity does not apply.

7.3 External Validity

External validity indicates whether it is possible to generalize the findings of the study.

Language choice. The study reported here is an extension of work initially performed with Pharo [13], but on a much smaller corpus. It was thus approriate for us to extend the experiment with more projects for that language. But the major purpose of this work was to show that the concept of rotten green tests also exists in other languages. For this purpose, we chose two other languages, Python and Java, which, according to GitHub[18], were respectively the second and third most used languages in 2019. We chose them because we have expertise in these two languages, whereas we have none in JavaScript (the most used language on GitHub). Moreover, JUnit4 in Java and pytest in Python are used in numerous projects, whereas in JavaScript it is harder to identify a dominant testing framework. Our implementations depend on test framework and version, so it was better to choose languages where almost all tests use the same framework. Finally, the purpose of the paper was to show

[18] https://octoverse.github.com/

that the concept of rotten green tests applies to languages other than Pharo. Finding rotten green tests in two common languages was sufficient for this purpose.

Project choice. Depending on the category, rotten green tests appear in 5 to 18 projects in Java and 12 to 30 in Python (see Table 1). Concerning *RQ2* and the possibility for a rotten test to hide a code bug, in the set of projects we studied we found one case in Pharo, one case in Python, but none in Java. Obviously, our choice of projects has an impact on these numbers. However, our focus was more on showing the existence of hidden bugs than on the quantity.

7.4 Reliability

Reliability is concerned with the extent to which the data and the analysis are dependent on the specific researchers.

To enable other researchers to conduct the same study, on one of the three chosen languages or on another, we describe a generic version of our approach in Section 3.3, and have given the details of the implementation for each language. We also detailed the method used to select the corpus of projects, and provide data files with the names of the projects in each language. The SHA of the commit is not specified, but all versions were current as of autumn 2019. Since the projects are on GitHub, it is possible to find the data in the history. We saved each project on disk to be sure that we always analyzed the same

version. Because of the number of projects, it is not possible to provide the full environment for each of them, but the results of our evaluation are available at https://github.com/rmod-team/2020-rotten-green-tests-experiment-data.

8 Discussion and Future Work

8.1 Location and Characterization of Helper Methods

Although it was not the focus of this article, we thought that it would be interesting to measure whether developers factor helper methods out of test classes and into superclasses. Table 2 measures the degree to which this occurs in the Java and Pharo projects that we analyzed. In Python, helper methods are often at module level rather than in the hierarchy of the test case that uses them, so we did not perform this analysis.

For Java and Pharo, we computed the *helper definition distance* — the number of levels of the class hierarchy between a call to a helper and its definition; Figure 4 illustrates the definition. A distance of 0 means that the helper is defined in the same class as the test, and a distance of 1 means that the helper is defined in the superclass of the class of the test, *etc. Other* means the helper methods are written in a class that does not belong to the class hierarchy of the test.

Across all the projects, we found a minimum distance of 0, *i.e.,* all projects using helpers define some of them directly in the test class that uses them. The maximum distance was 4 for Pharo projects, and 2 for Java.

L	Project	Other	0	1	2	3	4
J	flink-core	481	1612	2900	1456	0	0
J	optaplanner	1903	175	20	0	0	0
J	streamex	820	0	0	0	0	0
J	jeromq	42	415	210	0	0	0
J	bt	252	10	0	0	0	0
J	zemberek-nlp	0	90	0	0	0	0
J	spatial4j	32	12	34	0	0	0
J	joda-time	0	50	0	0	0	0
J	Sentinel	0	33	0	0	0	0
J	kuromoji	0	27	0	0	0	0
P	XML-Support-Pharo	0	1195	177	178	83	9
P	PetitParser	0	1254	24	18	0	0
P	PostgreSQLParser	0	3	243	243	0	0
P	MaterialDesignLite	0	9	75	359	16	0
P	multi-valued-dictionary	0	315	0	0	0	0
P	Seaside	0	227	13	16	3	0
P	ston	0	155	0	0	0	0
P	zinc	0	143	0	0	0	0
P	Exploratory-StudyInPharo	0	15	97	25	0	0
P	Famix	0	96	2	0	0	0

Table 2 For each project under analysis, the number of helper calls for the six distances of helper definition observed. The first column indicates the language: P for Pharo, J for Java.

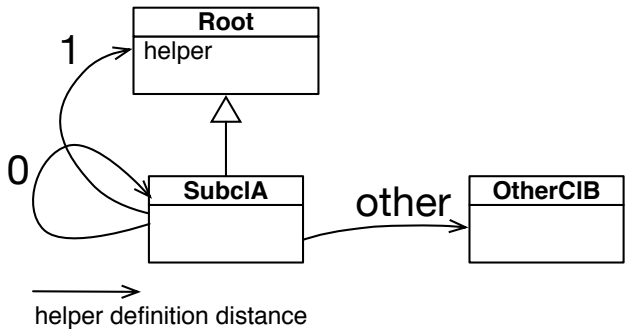

Fig. 4 Helper definition distance. A distance of 0 means that the helper is defined in the same class as the test, a distance of 1 means that the helper is defined in the superclass of the class of the test, and so on. *Other* means that the helper method is in a class outside the hierarchy of the test.

The distance of Java helpers varies widely between projects. There are projects whose test cases invoke helpers outside the hierarchy of the class's tests. For example, project Optaplanner contains 1903 tests invoking *external* helpers, such as PlannerAssert. However, other tests call helpers that are declared in the same class as the tests, or in their superclasses For example, project Flink-core has an abstract test class ParserTestBase; several concrete test classes, such as LongParserTest, extend it and use helpers defined in the abstract superclass.

The organization of helper methods has an impact on the abundance of rotten test cases. If a rotten helper method is declared outside the hierarchy (*Other*) or in a parent class (distance greater than 0), it is our intuition that the method is more likely to be reused, and will thus create more rotten green tests. This hypothesis explains the large number of rotten tests found in Java

projects such as Flink-core and Optaplanner. For instance, Optaplanner is the Java project with the largest number of rotten tests with 110 cases. Most of the rotten tests (103 cases) are from the context-dependent category. Those tests invoke a helper method (assertCode), which has a rotten assertion inside an if...else. For future work we plan to study the correlation between helper distance and implication in rotten tests.

8.2 Guidelines to Avoid and Correct Rotten Tests

As we have shown, rotten green tests are caused by potentially faulty conditional logic in tests that have branches. The simplest way to avoid them is to remove conditional logic from tests. However, the reality of testing does not allow test engineers to apply that solution in all cases. We can draw insight from Meszaros' work on the *Conditional Test Logic* smell [26], which is defined as *a test containing code that may or may not be executed.* For example, Listing 11's conditional asserts (see page 31) that depend on the Python version might seem to be a useful practice. The cause of the smell is a so-called *Flexible Test* [26], for which solutions include refactoring tests into separate modules, one of which could be injected into a testing framework depending on the environment (*i.e.,* Python 2 or 3). Other causes of this smell include managing complex fixtures (set-up and teardown), trying to test complex (polymorphic) objects, trying to test collections of objects, and dealing with invalid data from the system under test. For each of these causes, Meszaros proposes possible solution patterns, including *Guard Assertion*, *Equality Assertion*, *Test Utility*

Method, and *Custom Assertion* with *Custom Assertion Tests*. Indeed, all of these patterns could be applied to the problem of avoiding rotten green tests.

From the specific analysis we performed on Pharo, Java, and Python, we observed that many of the rotten tests can be systematically corrected. We believe that a classification of the errors observed in rotten tests could lead to the creation of guidelines for automatic or semi-automatic repair of rotten tests, potentially based on Meszaros' patterns.

8.3 Anti-patterns: Misuse of testing frameworks

The results of Table 1 show that several projects contain multiple instances of *missed fail* and *missed skip* tests. As discussed in the definition of those categories, they can be viewed as a misuse of the testing frameworks. Missed fail is when a test calls assertTrue(**false**) (*i.e.,* **false** is a literal argument) instead of fail()[19]; missed skip is when a test uses conditional logic, *e.g.,* a conditional return, rather than a feature such as assumeTrue(boolean) in Java or pytest.mark.skipif()[20] in Python. Based on the results of our study, we consider these two categories to be potential test anti-patterns (*i.e.,* test smells) and plan to investigate them in future work.

8.4 Future work

From the discussion above, here is a summary of future work:

[19] See the discussion on StackOverflow at https://stackoverflow.com/q/12939362/1168342

[20] See https://docs.pytest.org/en/reorganize-docs/new-docs/user/skipping.html.

- Study the relationship between method distance and implication in rotten green tests.
- Identify properties of source code that reveal rotten green tests.
- Provide automated strategies to correct rotten green tests, using guidance from the literature including the *Conditional Test Logic* smell [26].
- Evaluate how these properties and strategies apply to other programming languages.
- Investigate the proposed *missed fail* and *missed skip* anti-patterns.

9 Related Work

Software testing is an active area of research; many researchers have looked at improving the quality of tests. The only work of which we are aware concerning rotten green tests is our own [12, 13, 25].

Test analysis. Herzig et al. [21] present an approach based on machine learning to detect false test alarms. These are integration tests that are directly linked to code defects but fail due to external factors such as hardware failing to give access to a test resource. Identifying false test alarms is key since they demand the attention of engineers, and require manual checking to determine that they are false alarms.

Vera Perez et al. [38] present a novel analysis of pseudo-tested methods. These are methods that are covered by tests, yet no test case fails when the method body is removed. This intriguing concept was named in 2016 by Nie-

dermayr et al., who showed that such methods are systematically present, even in well-tested projects with high statement coverage [28]. Pseudo-tested methods are indirectly related to rotten tests: one possible cause of a pseudo-tested method is that all the tests of that method are rotten. The existence of pseudo-tested methods shows that it is important to improve tool support for testing.

Huo et al. [22] use taint analysis to identify brittle or unused test inputs. They monitor the flow of controlled and uncontrolled inputs along data and control dependencies. Our approach focuses on monitoring assertion execution; from that perspective it is complementary. Mockus et al. [27] present an approach to determine test effectiveness. Their analysis compares test coverage prior to a release with the number of reported failures after that same release.

We agree with Schuler and Zeller that simple coverage is not enough, especially in presence of dynamic dead code [32]. They propose instead to use *checked coverage*, and aim to improve code coverage by identifying program elements that are not in the dynamic backward slice of any assertion. Thus their work, like ours, identifies tests that make no assertions. Our work does not seek to improve coverage, and consequently we do not look for statements that are not covered by tests. Instead, we seek to ensure that when a test is green, some assertions have been made about the code covered by that test. Our analysis is less general, but also simpler to implement and faster to run.

Mutation testing. Mutation testing is one of the earliest techniques used to improve test quality and robustness [14]. Several researchers have used muta-

tion testing to improve branch coverage [24]. Tillmann et al. [35] use symbolic execution to find inputs for parametrized unit tests that achieve high code coverage. They turn existing tests into parametrized tests, and generate entirely new parametrized tests that describe the behavior of an existing implementation. Baudry et al. [1] present a bacteriological approach to mutation testing. Baudry et al. [2] also worked on improving a test-for-diagnosis criterion: they propose a new attribute, the Dynamic Basic Block, to improve the location of faults. For fault localization, the usual assumption is that test cases satisfying a chosen criterion for test adequacy are sufficient to perform diagnosis. This assumption is verified neither by specific experiments nor by intuitive considerations.

Repairing broken tests. Daniel et al. [11] present ReAssert, a tool that repairs broken tests. They define broken tests as tests that turn red because the domain code is changed. They propose various repair strategies, such as replacing asserted values, inverting relational operators, and replacing some common method calls. By definition, rotten green tests are not broken tests because they are green.

Test smells. Several researchers have focused on test smells: our approach is related to such work since rotten tests can be seen as a test smell, although none of the existing catalogs mentions them. Deursen et al. [15] present a list of "bad test smells" and their associated cures, but do not mention rotten green tests. Van Rompaey et al. [36, 37] propose a metric-based heuristic to identify

two test smells (General Fixture and Eager Test). They present TestQ, a tool that uses static analysis to detect test smells [8]. Bavota et al. [3] present an empirical analysis to assess the presence of test smells and their impact on software maintenance. Reichhart et al. [30] propose TestLint, a rule-based tool to detect static and dynamic test smells. Based on a large corpus of tests and a literature analysis, they collected and proposed a list of 27 test smells. Some proposed smells are studied. For example, their analysis identifies commented-out assertions; they check if uncommenting them leads to valid tests. Rotten tests were not part of their list. Rotten tests often exhibit Conditional Logic smells [26, 30]. In this paper we describe how such a smell can be detected automatically. More recently, Bowes et al. [7] identified 15 principles that they claim capture the essential goals of testing. Eight of their principles are not covered by existing test smells. They proposed some metrics to measure the degree to which tests follow their principles.

Silva et al. [34] selected 14 widely studied test smells from the literature and ran experiments with professional developers. Their results indicate that experienced developers introduce test smells during their daily programming tasks, even when they are using standardized practices. Another result is that developers' professional experience cannot be considered as a root cause for the insertion of test smells in test code.

Meszaros [26] documented various test code smells, but none specifically addresses the notion of rotten green tests. As discussed above, the *Conditional Test Logic* smell is related; it is described by the phrase: "When tests have mul-

tiple execution paths, we cannot be sure exactly how the test will execute in a specific case." This smell includes loops, because they create multiple execution paths. This smell is potentially a factor in all categories of rotten green tests. Meszaros identifies the causes of this smell as including if statements used to steer execution to a fail statement, loops that verify the contents of collections, and conditional logic in a test fixtures intended to allow a single test to verify several different cases. Solutions proposed by Meszaros include using a *Guard Assertion* to cause a test to fail before reaching code that should not execute, and moving loops to *Custom Assertions* with *Custom Assertion Tests*. Meszaros also recommends writing unit tests for *Test Utility Methods* when they contain "complex algorithms." This is related to our work, since we consider helper methods as part of identifying rotten green tests. Note that, in the context of studying rotten green tests, it is Conditional Test Logic that affects the execution (or not) of assertions.

Test effectiveness. A large body of research has been carried out to assess the effectiveness of tests at detecting faults. Mockus et al. [27] conducted a multiple-case study on two dissimilar industrial projects. They found that, in both projects, an increase in test coverage is associated with a decrease in field-reported problems when adjusted for the number of pre-release changes. Inozemtseva and Holmes [23] conducted a large and intensive study showing that code coverage is not strongly correlated with test suite effectiveness. Because of this, they suggest that coverage, while useful for identifying under-tested parts of a program, should not be used as a quality property. Gligoric

et al. [20] perform a study showing that branch coverage and intra-procedural acyclic path coverage are the criteria that perform the best when comparing inadequate test suites.

Test selection. Other work focuses on the selection of the tests to be run. For example, when a change is made to the software, it is desirable to re-run those tests that are most likely to be invalidated by the change. Beszedes et al. [4] propose using code coverage for test selection, to maximize the test surface. Blondeau et al. [6] analyze the problem of test selection surfaces in an industrial context.

System robustness For test input dependability, using a different approach, Poulding and Feldt [29] increase the quality of unit tests. They automatically generate new invalid and atypical inputs for application tests. They assess the impact of the choice model on the input variability, using a global probability distribution over all the inputs that could be emitted. Rotten tests are not related to the quality of invalid or atypical inputs. They are linked to unsatisfied test context and conditional logic that confused developers. In the spirit of JCrasher [10] and system robustness, Shahrokni et al. [33] present RobusTest, a framework to generate automatically tests for timing issues.

Broken test sorting. Often, unit test frameworks present failed tests in an arbitrary order, but developers want to focus on the most specific ones first. Gaelli et al. propose a partial order of unit tests corresponding to a coverage lattice of sets of covered method signatures [19]. When several unit tests in

this coverage lattice fail, the tool guides the developer to the test invoking the smallest set of methods.

10 Conclusion

This paper presents an extended study of *rotten green tests*: tests that were intended by their designer to execute some assertions, but that do not actually do so. Such tests are insidious because they pass, *and they contain assertions*; they therefore give the *impression* that some useful property is being validated. In fact, rotten green tests guarantee nothing: they give a false impression of confidence. Rotten green tests can be seen as a new kind of test smell [15, 30].

The original *rotten green tests* study [13] identified rotten green tests in Pharo [5], a dynamically typed language inspired by Smalltalk. This study extends the original one by using a much larger corpus of projects, and adding two mainstream programming languages with different programming idioms and characteristics: Python and Java. We generalized the approach for detecting rotten green tests to make it applicable to all three languages, and constructed an implementation for each language using a combination of a simple static analysis and dynamic monitoring of method execution.

We analyzed a total of 276 projects, and found that 85 of them present one or more rotten green tests, accounting for a total of 665 rotten green tests. Our findings show that the categories of rotten green tests we found in the original study are also present in other languages. We see that most of the rotten green

tests we detected are caused by context-dependent assertions, although there exist cases of fully rotten tests that hide application bugs.

As developers, we want to trust our test framework. When it says that a test containing some assertions is green, this should mean it has really executed the assertions in that test, and that they really do hold. We hope that in the near future the maintainers of various testing frameworks consider the detection of rotten green tests.

References

1. Baudry B, Fleurey F, Jézéquel JM, Traon YL (2005) Automatic test case optimization: A bacteriologic algorithm. IEEE Software 22(2):76–82
2. Baudry B, Fleurey F, Traon YL (2006) Improving test suites for efficient fault localization. In: ICSE '06: Proceeding of the 28th international conference on Software engineering, ACM Press, New York, NY, USA, pp 82–91, DOI 10.1145/1134285.1134299
3. Bavota G, Qusef A, Oliveto R, Lucia AD, Binkley D (2012) An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In: International Conference on Software Maintenance (ICSM), IEEE, pp 56–65, DOI 10.1109/ICSM.2012.6405253
4. Beszedes A, Gergely T, Schrettner L, Jasz J, Lango L, Gyimothy T (2012) Code Coverage-based Regression Test Selection and Prioritization in WebKit. In: Software Maintenance (ICSM), 2012 28th IEEE International Conference on, pp 46–55, DOI 10.1109/ICSM.2012.6405252
5. Black AP, Ducasse S, Nierstrasz O, Pollet D, Cassou D, Denker M (2009) Pharo by Example. Square Bracket Associates, Kehrsatz, Switzerland, URL http://books.pharo.org
6. Blondeau V, Etien A, Anquetil N, Cresson S, Croisy P, Ducasse S (2016) Test case selection in industry: An analysis of issues related to static approaches. Software Quality Journal pp 1–35, DOI 10.1007/s11219-016-9328-4

7. Bowes D, Tracy H, Petrié J, Shippey T, Turhan B (2017) How good are my tests? In: Workshop on Emerging Trends in Software Metrics (WETSoM), IEEE/ACM
8. Breugelmans M, Van Rompaey B (2008) TestQ: Exploring structural and maintenance characteristics of unit test suites. In: International Workshop on Advanced Software Development Tools and Techniques (WASDeTT)
9. Costiou S, Aranega V, Denker M (2020) Sub-method, partial behavioral reflection with Reflectivity: Looking back on 10 years of use. The Art, Science, and Engineering of Programming 4(3), DOI 10.22152/programming-journal.org/2020/4/5
10. Csallner C, Smaragdakis Y (2004) JCrasher: an automatic robust tester for Java. Software: Practice and Experience 43
11. Daniel B, Dig D, Gvero T, Jagannath V, Jiaa J, Mitchell D, Nogiec J, Tan SH, Marinov D (2011) Reassert: A tool for repairing broken unit tests. In: Proceedings of the 33rd International Conference on Software Engineering, ACM, New York, NY, USA, ICSE '11, pp 1010–1012, DOI 10.1145/1985793.1985978, URL http://doi.acm.org/10.1145/1985793.1985978
12. Delplanque J, Ducasse S, Black AP, Polito G (2018) Rotten green tests: a first analysis. Tech. rep., Inria
13. Delplanque J, Ducasse S, Black AP, Polito G, Etien A (2019) Rotten green tests. In: 2019 IEEE/ACM 41st Int. Conf. on Software Engineering (ICSE), pp 500–511, DOI 10.1109/ICSE.2019.00062

14. DeMillo RA, Lipton RJ, Sayward FG (1978) Hints on test data selection: Help for the practicing programmer. Computer 11(4):34–41, DOI 10.1109/C-M.1978.218136, URL http://dx.doi.org/10.1109/C-M.1978.218136
15. van Deursen A, Moonen L, van den Bergh A, Kok G (2001) Refactoring test code. In: Marchesi M (ed) Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes (XP2001), University of Cagliari, pp 92–95
16. Ducasse S, Nierstrasz O, Schärli N, Wuyts R, Black AP (2006) Traits: A mechanism for fine-grained reuse. ACM Transactions on Programming Languages and Systems (TOPLAS) 28(2):331–388, DOI 10.1145/1119479.1119483
17. Ducasse S, Pollet D, Bergel A, Cassou D (2009) Reusing and composing tests with traits. In: TOOLS'09: Proceedings of the 47th International Conference on Objects, Models, Components, Patterns, Zurich, Switzerland, pp 252–271
18. Dustin E, Rashka J, Paul J (1999) Automated Software Testing: Introduction, Management, and Performance. Addison-Wesley Professional
19. Gaelli M, Lanza M, Nierstrasz O, Wuyts R (2004) Ordering broken unit tests for focused debugging. In: 20th International Conference on Software Maintenance (ICSM 2004), pp 114–123, DOI 10.1109/ICSM.2004.1357796, URL http://scg.unibe.ch/archive/papers/Gael04aOrderingBrokenUnitTestsForFocusedDebugging.pdf
20. Gligoric M, Groce A, Zhang C, Sharma R, Alipour MA, Marinov D (2013) Comparing non-adequate test suites using coverage criteria. In: Interna-

tional Symposium on Software Testing and Analysis

21. Herzig K, Nagappan N (2015) Empirically detecting false test alarms using association rules. In: International Conference on Software Engineering
22. Huo C, Clause J (2014) Improving oracle quality by detecting brittle assertions and unused inputs in tests. In: Foundations on Software Engineering
23. Inozemtseva L, Holmes R (2014) Coverage is not strongly correlated with test suite effectiveness. In: International Conference on Software Engineering
24. Lingampally R, Gupta A, Jalote P (2007) A multipurpose code coverage tool for Java. In: System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on, pp 261b–261b, DOI 10.1109/HICSS.2007.24
25. Martinez M, Etien A, Ducasse S, Fuhrman C (2020) Rtj: a Java framework for detecting and refactoring rotten green test cases. In: IEEE/ACM 42nd Int. Conf. on Software Engineering: Companion Proceedings (ICSE '20 Companion), 5–11 Oct. 2020, Seoul, Republic of Korea, pp 69–72, DOI 10.1145/3377812.3382151
26. Meszaros G (2007) XUnit Test Patterns – Refactoring Test Code. Addison Wesley
27. Mockus A, Nagappan N, Dinh-Trong TT (2009) Test coverage and post-verification defects: A multiple case study. In: Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement, IEEE Computer Society, Washington, DC, USA, ESEM '09, pp 291–301, DOI 10.1109/ESEM.2009.5315981, URL http://dx.doi.org/10.1109/ESEM.

2009.5315981

28. Niedermayr R, Juergens E, Wagne S (2016) Will my tests tell me if I break this code? In: International Workshop on Continuous Software Evolution and Delivery, ACM Press, pp 23–29
29. Poulding SM, Feldt R (2017) Generating controllably invalid and atypical inputs for robustness testing. In: IEEE International Conference on Software Testing, Verification and Validation Workshops, pp 81–84, DOI 10.1109/ICSTW.2017.21, URL https://doi.org/10.1109/ICSTW.2017.21
30. Reichhart S, Gîrba T, Ducasse S (2007) Rule-based assessment of test quality. In: Journal of Object Technology, Special Issue. Proceedings of TOOLS Europe 2007, vol 6/9, pp 231–251
31. Runeson P, Höst M (2009) Guidelines for Conducting and Reporting Case Study Research in Software Engineering. Empirical software engineering 14(2):131–164
32. Schuler D, Zeller A (2013) Checked coverage: an indicator for oracle quality. Software testing, verification and reliability 23:531–551, DOI 0.1002/stvr.1497
33. Shahrokni A, Feldt R (2011) Robustest: Towards a framework for automated testing of robustness in software. In: International Conference on Advances in System Testing and Validation LifeCycle
34. Silva Junior N, Rocha L, Martins LA, Machado I (2020) A survey on test practitioners' awareness of test smells. arXiv preprint arXiv:200305613

35. Tillmann N, Schulte W (2005) Parameterized unit tests. In: ESEC/SIGSOFT FSE, pp 253–262, URL ftp://ftp.research.microsoft.com/pub/tr/TR-2005-64.pdf
36. Van Rompaey B, Du Bois B, Demeyer S (2006) Characterizing the relative significance of a test smell. Proceedings of ICSM 2006 0:391–400, DOI 10.1109/ICSM.2006.18
37. Van Rompaey B, Du Bois B, Demeyer S (2006) Improving test code reviews with metrics: a pilot study. Tech. rep., Lab on Re-engineering, University of Antwerp
38. Vera-Perez O, Danglot B, Monperrus M, Baudry B (2018) A comprehensive study of pseudo-tested methods. CoRR abs/1807.05030, URL https://arxiv.org/abs/1807.05030, 1807.05030