Rotten Green Tests
Julien Delplanque, Stéphane Ducasse, Guillermo Polito, Andrew Black, Anne Etien

To cite this version:
Julien Delplanque, Stéphane Ducasse, Guillermo Polito, Andrew Black, Anne Etien. Rotten Green Tests. ICSE 2019 - International Conference on Software Engineering, May 2019, Montréal, Canada. hal-02002346v2

HAL Id: hal-02002346
https://hal.inria.fr/hal-02002346v2
Submitted on 22 May 2019

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.
Rotten Green Tests

Julien Delplanque*, Stéphane Ducasse†, Guillermo Polito*, Andrew P. Black§‡ and Anne Etien*

*Univ. Lille, CNRS, Centrale Lille, Inria, UMR 9189 - CRISTAL, F-59000 Lille, France
†RMOD - Inria Lille, France
§Dept of Computer Science, Portland State University, Oregon, USA
*†{firstname}.{lastname}@inria.fr §apblack@pdx.edu

Abstract—Unit tests are a tenet of agile programming methodologies, and are widely used to improve code quality and prevent code regression. A green (passing) test is usually taken as a robust sign that the code under test is valid. However, some green tests contain assertions that are never executed. We call such tests Rotten Green Tests.

Rotten Green Tests represent a case worse than a broken test: they report that the code under test is valid, but in fact do not test that validity. We describe an approach to identify rotten green tests by combining simple static and dynamic call-site analyses. Our approach takes into account test helper methods, inherited helpers, and trait compositions, and has been implemented in a tool called DrTest. DrTest reports no false negatives, yet it still reports some false positives due to conditional use or multiple test contexts. Using DrTest we conducted an empirical evaluation of 19,905 real test cases in mature projects of the Pharo ecosystem. The results of the evaluation show that the tool is effective; it detected 294 tests as rotten—green tests that contain assertions that are not executed. Some rotten tests have been “sleeping” in Pharo for at least 5 years.

I. INTRODUCTION

Agile methodologies such as Extreme Programming [5] promote Unit Testing [29] as a key tenet of the software development process. Executing a test suite after each change to the software helps to ensure that new functionality works, and that the old functionality remains working, that is, it helps to avoid software regressions [1].

Tests are based on the execution of assertions that check that the system under test satisfies some property, for example, that a method returns a certain value, or that certain data is written to a stream. Developers value “green tests”, i.e., tests that are passing, because they increase the developers’ confidence that the software is working as expected.

The software engineering research community has developed many techniques to help assess the quality of tests. Some approaches are based on mutation testing [2, 3, 15, 26]; others on code coverage [6, 30, 41] or on repairing tests [14]. Several researchers [4, 10, 17, 29, 34, 42] have worked on “test smells” (analogous to Fowler et al.’s Code Smells [20]). Huo et al. [24] present OraclePolish, which identifies brittle or unused test inputs. Pinto et al. [32] performed an analysis of the evolution of test suites; they mention repaired, added and removed tests.

Our concern in this work is with 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 bogus feeling of confidence. Rotten green tests can be seen as a new kind of test smell.

Our approach to identifying rotten green tests uses a combination of a simple static analysis and dynamic monitoring of method execution at the granularity of individual call-sites; it can determine whether or not a test is rotten, even in presence of helper methods and trait compositions [18, 19]. Our approach does not report false negatives, but can report false positives in the case of conditional tests, and tests that are reused in multiple contexts. While our implementation and validation are in Pharo [7], rotten green tests are not specific to Pharo; we encourage replication of our approach in other languages.

The contributions of this work are:

• the identification of rotten “green” unit tests as passing tests that contain assertions that are not executed, and which therefore give the developers false confidence;
• a simple and precise approach that finds rotten green tests by combining static and dynamic call-site analyses; and
• a detailed analysis of the results of applying this approach to several large systems containing a total of 19,905 tests.

II. PHARO SYNTAX IN A NUTSHELL

We use plain Pharo code [7] in this article rather than attempting to translate our real examples into another language. Pharo, as a Smalltalk descendant, is a conceptual subset of Java: it can be considered to be a Java without a static type system that makes heavy use of closures. Everything in Pharo is an object, and every object is an instance of a class. Each class inherits from a single superclass. All methods are public virtual; the fields in an object are accessible only to that object’s methods, and are thus more protected that fields in Java. In addition, Pharo classes can be composed from traits — reusable groups of method definitions.
Fields and local variables are read by using their name, and written using `:=` for assignment. Method invocations use space (rather than a dot), and colon introduces an argument, so

```
receiver methodN1: arg1 n2: arg2
```

is equivalent to the Java syntax:

```
receiver.methodNin2{arg1, arg2}
```

Hence,

```
self assert: s size equals: 1
```

corresponds to

```
this.assertEquals(s.size(), 1)
```

In Pharo, periods separate statements and are the equivalent of Java semi-colons. A caret `^` means `return` from the current method; if there is no explicit return, the receiving object is returned by default. Thus, the following Pharo statement

```
^ weather isRaining
  ifTrue: [ self takeUmbrella ]
  ifFalse: [ self takeSunglasses ]
```

is equivalent to this Java statement:

```
return ( if (weather.isRaining())
  { this.takeUmbrella() }
  else { this.takeSunglasses() })
```

This code snippet also illustrates the use of messages and lexical closures, delimited by `[ and ]`, to implement control flow, such as conditionals and loops; this is ubiquitous in Pharo. Parameters to a closure are introduced by a colon and terminated by a vertical bar `|`, so `[^ x | x + 2]` is a closure that adds 2 to its argument. Closures are executed by sending them the message `value:`, so `[^ x | x + 2].value: 33` returns 35.

Strings are delimited by single quotes, while comments use double quotes: "this is a comment". The notation `ClassName » methodName` is not part of Pharo itself, but is used to designate a method when writing about Pharo; we will use it here.

### III. The Problem of Rotten Green Tests

Before defining rotten green tests, we first describe the basics of unit testing, and then briefly explain “Smoke Tests”, to help distinguish them from the topic of this article.

#### A. Unit tests

Unit tests are commonly composed of a test `fixture` (which sets up the system to be tested), one or more `stimuli` (which exercise the component under test), and one or more `assertions` that verify some expected property [7, 28]. Listing 1 shows a trivial `SUnit` test for a set.

In the example, the `fixture` is the code that declares and initializes `s` to contain 1; here the fixture is inline, but it can also be factored-out into a `setUp` method. The `stimulus` is the second addition of 1 to `s`; the `assertions` then verify the property that `s` contains 1 just once.

Provided that all of the assertions are true, this test will pass; we say that it is “green”. If a false assertion is executed, for example, if the set does not detect the duplicate and its size is 2, the test will fail: it will be “yellow”. If an error occurs during the running of the test, for example, if `Set new` signals an exception, then the test will be “red”.

#### B. Smoke Tests

It is common practice to use unit testing frameworks to execute so-called “smoke tests”, whose purpose is to check that the feature under test can be run without emitting “blue smoke” — that is, that the test ran without raising an unexpected exception [44]. Listing 2 is an example of such a test.

```
SetTest » testSetAddSmokeTest
| s |
s := Set new. "Fixture"
s add: 1. "Stimulus"
s add: 1. "Stimulus"
self assert: s size equals: 1. "Assertions"
sself assert: (s includes: 1).
```

Listing 1: the parts of a unit test.

Listing 2: a smoke test.

In its simplest form, a smoke test may contain no assertions at all, as in this illustration. This is the way that we use the term “smoke test” in the remainder of this article.

Smoke tests are useful because, if they are green, they provide a fast but cursory check that the feature concerned can be considered for further testing. Conversely, if a smoke test is red, there is a serious issue that should be addressed rapidly. Smoke tests are not the concern of this article; nothing that follows should be construed as advocating either for or against the use of smoke tests. Nevertheless, we do need to distinguish a smoke test that by design contains no assertions, from a rotten green test, which by accident executes no assertions.

#### C. 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 test that does contain a valid fixture, stimulus and assertion, but nevertheless does not `execute` any assertion. Listing 3 shows a real example, from Pharo issue 7478.
At first glance, this looks like a fine test of the collection printOn:delimeter: method. Lines 3 and 4 create a fixture—in this case they set up aStream. Then comes the stimulus: we send the message printOn: aStream delimiter: '; ', to the collection resulting from self nonEmpty, causing it (we hope) to write its elements to aStream, delimited by commas. The remainder of the test is intended to make assertions. Lines 6–10 look as though they are parsing the contents of aStream and asserting that the elements found on it are equal to those in the original collection.

This test is passing, so we know that everything is OK, right? Wrong! The programmer who wrote this test misunderstood the way that streams work. The string in the variable result can never be modified by writing to aStream. Indeed, the name result is misleading, because result is initialized to the empty string and never changes. Consequently, result findBetweenSubstrings: ', ', will answer an empty collection, and the withIndexDo: block—which contains the only assertion—will never be executed. We could put any assertion into this block, and the test would still be green.

We believe that such a test is worse than no test at all. First, the rotten test does not guarantee any property of the method under test. Second, it wastes time—remember that we want our tests to run quickly. Third, in the case of “no test at all”, test coverage statistics would reveal that the printOn:delimeter: method is not being exercised, and the developers would at least be aware that their testing is inadequate. Instead, a rotten test exercises the method without actually testing it.

Rotten green tests give developers a false sense of security: coverage may be good, and the tests may be green, but in fact no properties are being checked (other than the property that the code runs without signaling an error). Moreover, when developers look at such a test superficially, they will probably not spot that there is a problem.

The example in Listing 3 was first reported as a bug in February 2013. However, because the test was green, the bug was easy to overlook, and the bug report was closed without any action being taken. As of the end of 2017, this rotten green test was still present in Pharo.

IV. IDENTIFYING ROTTEN GREEN TESTS

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 makes no assertions, but this won’t let us distinguish smoke tests from rotten green tests. Neither can we assume that any test that contains no assertions is a smoke test: many tests make assertions indirectly through the use of helper methods (see Section IV-A). Tests that use assertions in helper methods are not smoke tests, even though they contain no assertions.

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. In Pharo, there are 34 assertion primitives implemented in the asserting protocol of class Test_Assert (for example, assert: , assert:equals: , and deny:). We do not consider fail: , fail, and signalFailure: to be assertion primitives because these methods are designed to be invoked conditionally. Similarly, we exclude skip and skip: , which allow one to skip the test without signalling a failure.

- A test method is a method identified as containing a test by the unit-testing framework. In Pharo, test methods are zero-argument methods that are defined in a subclass of TestCase, and whose names start with ‘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 that is not a test method. Developers frequently write application-specific test helper methods, as we discuss in Section IV-A.

A. Helper methods

It is common practice for developers to factor-out assertions into helper methods. Listing 4 shows an example for numerical checks. 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.

```
TPrintOnSequencedTest = testPrintOnDelimiter
| aStream result allElementsAsString |
result := "."

aStream := ReadWriteStream on: result.
self nonEmpty printOn: aStream delimited: '; ',
allElementsAsString := result findBetweenSubstrings: '; ',
allElementsAsString withIndexDo: [:el :i |
    self
    assert: el
equals: (self nonEmpty at:i) asString)
```

Listing 3: a rotten green test.

```
NumericalTests » assert: actual isRoughly: desired within: eps
self
assert: (actual - desired) abs <= eps
description: [ ’actual result ’, actual ,
’is not even roughly equal to’ , desired ]
```

Listing 4: a helper method for approximate arithmetic

For a real-world example of a helper method, we turn to the Pillar editing platform [12]. Pillar’s test suites use helper methods such as assertWriting:includesText:; which is defined in the superclass of Pillar’s test classes, PRDocumentWriterTest. This method factors out the writing of an HTML element and the checking of the emitted HTML. Listing 5 shows its definition, and a sample of its use.

Like a rotten test, a helper method might fail to make an assertion in some or all situations. In other words: helper methods might be rotten too. Any approach to detecting rotten green tests should also detect rotten helper methods. Moreover, since the action of a rotten method may depend on the context (e.g., the test fixture, and the arguments to the helper), we
need to record the specific test that exposes the issue, because a helper might work in one context, but be rotten in another.

B. Classifying Tests

There are three situations that a rotten test analysis should identify.

Good test. A test passes and contains some assertions (either directly, or indirectly though helper methods), and all these assertions are executed: the test is good.

Rotten test. A test passes, and contains assertions (either directly, or indirectly), but at least one of its assertion is not executed: the test, or the helper method, is rotten.

Smoke test. A test contains no assertions (either directly, or indirectly); it is a smoke test.

Note that distinguishing between good and rotten requires some dynamic analysis, because we need to ascertain whether an assertion is executed. In contrast, distinguishing between rotten and smoke requires some static analysis, because we need to ascertain whether the test contains assertions.

Listing 6 shows why detecting rotten tests requires an analysis with call-site granularity. testABC is a test method, and helper and secondHelper are helper methods (because helper invokes secondHelper, and secondHelper invokes an assertion primitive). On line 4, testABC contains a valid assertion self assert: true, which is always executed. In contrast, the call to method helper on line 3 will not be executed; as a consequence the assertion in secondHelper will not be executed either. Thus, this test contains (indirectly) an assertion that is not executed: this makes it rotten by our definition. To detect this, an analysis must distinguish between the two assertion call sites.

C. Combining Static and Dynamic Analyses

Our analysis performs the following steps.

Step 1: Identification of assertion primitives. We build the set of assertion primitives by manually tagging all the methods of the unit test framework that make assertions. This set depends only on the test framework.

Step 2: Identification of helper methods. To compute the set of helper methods, we statically build a call tree of all the self and super-sends from the test method under analysis. Building this tree takes care of chains of method invocations as far as the assertion primitives. The tree is then pruned to remove all branches that do not lead to an assertion primitive. All methods remaining in the pruned tree, except the root and the leaves, are helper methods.

Step 3: Call-site instrumentation. We instrument all sites in the test method and helper methods from which assertion primitives and helpers are called. That is to say, we set up a mechanism that allows us to determine whether or not, after the execution of a method, each call site of an assertion primitive or a helper has been executed. If a call site has not been executed, the mechanism gives us the corresponding AST node.

Step 4: Test execution. We execute each test method (including inherited test methods) one at a time, while monitoring:

a) the outcome of the test (pass, fail, or error), and

b) whether each assertion primitive and helper call site has been executed.

Because we are looking for rotten green tests, we consider only passing tests. A method (test or helper) is considered to be rotten when it contains a call site for an assertion primitive or a helper, but this assertion or helper was not invoked.

Step 5: Report generation. DrTest’s final report has to take into account the way that methods are reused in the test hierarchy. A test method may be defined in a superclass and executed in a subclass. In general, the test fixture, and thus the meaning of the test, will be different in each place in which it is dynamically used, so we must report the class of the test as well as the method. Helper methods are designed to be used by many test methods; if a helper method is rotten, the test invoking it should be reported, so that the programmer can understand the scenario in which the helper fails to make an assertion.

The dynamic analysis that we use has call site granularity: we know whether each call site has been executed or not. Therefore we can identify a method in which one assertion primitive was executed while another one was not. This is why we do not have false negatives.

D. Example Explained

To explain how the approach works, we use the examples from Listing 6 and 7. In Listing 6, the call on line 4 is executed. However, the call self helper on line 3 is not executed. Because method helper has been identified as a helper method by the static analysis, testABC is determined to be rotten.
In Listing 7, the assertion in line 4 is executed and the method `helper` is executed. However, the assertion in `secondHelper` is not executed because the condition at line 7 is always false. Therefore, `helper` is determined to be rotten, as is `testDEF`.

```
RottenTest = testDEF
"Test method"
self badHelper.
self assert: true.
RottenTest = badHelper
"Indirect helper"
false ifTrue: [ self secondHelper ]
RottenTest = secondHelper
"Direct helper"
self assert: x
```

Listing 7: a rotten helper method; badHelper makes no assertions

V. VALIDATION AND RESULTS

In this section we describe what we found when we applied our approach for detecting rotten tests to some real software projects. Section V-A characterizes the projects whose tests we analysed. Section V-B presents a classification of the rotten tests we found, and Section V-C summarises our results. Finally, Section V-D describes the results from each project, and examines some of the rotten tests that we found.

A. Characterizing the projects under analysis

Pharo is an open-source language derived from Smalltalk with a growing community and a large number of mature projects. We have run DrTest on 17 Pharo subsystems.

Table I characterizes the analysed projects. For each project we show a short description of its purpose, the number of packages, the number of classes in all these packages, the number of test classes, the number of tests, the number of helpers and the number of rotten tests in each of four categories, which will be described in the next subsection.

Note that first three categories are not exclusive, so a rotten test can appear in more than one of them.

We have chosen these projects because of their heterogeneity. They are quite different in terms of purpose, packages, classes, tests and use of helpers.

B. Rotten test categorisation

Not all rotten tests are equal. Some follow certain patterns, and are simple to address, while others are complex and hard to characterise. We originally planned to classify rotten tests based on structural properties such as the use of helper methods and hierarchy. However, such criteria turned out not to be interesting because they depend on a developer choice, and are not related to the cause of the rotten test.

Instead, we performed a manual analysis and assessment of each of the rotten tests identified by DrTest (see Section V-D). The results of this analysis led to the emergence of four categories: missed fail, missed skip, context dependent assertions, and fully rotten tests.

While our approach definitively identified assertions that were not executed, we interpret the two first categories (missed fail and missed skip) as false positives because they show that the developer misused the assertion primitives provided by the test framework, and not that the test was misleading.

We present each category and discuss our findings.

a) Missed fail: This category contains tests where the developer passed false to the `assert` primitive to force the test to fail. This is illustrated in Listing 8, line 6, where the `detectNilNone` iterator is used to find an element satisfying the predicate passed as first argument. When there is no element satisfying this predicate, the second argument (another closure) is executed. So the test is written to fail if no suitable object is found in the collection.

To make this test fail, the developer used `self assert: false` instead of `self assert: true`; this is a misuse of the testing framework. Our approach classifies this test as rotten because the assertion

<table>
<thead>
<tr>
<th>Project</th>
<th>Description</th>
<th>packages</th>
<th>classes</th>
<th>tests</th>
<th>test classes</th>
<th>helpers</th>
<th>missed fail</th>
<th>missed skip</th>
<th>context dependent</th>
<th>fully rotten</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler</td>
<td>AST model and compiler of Pharo.</td>
<td>6</td>
<td>232</td>
<td>51</td>
<td>859</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Aconcagua</td>
<td>Model representing measures.</td>
<td>2</td>
<td>84</td>
<td>27</td>
<td>661</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Buoy</td>
<td>Various package extensions</td>
<td>12</td>
<td>51</td>
<td>19</td>
<td>185</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Calypso</td>
<td>Pharo IDE.</td>
<td>58</td>
<td>705</td>
<td>157</td>
<td>2692</td>
<td>4</td>
<td>88</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Collections</td>
<td>Pharo collection library.</td>
<td>16</td>
<td>222</td>
<td>59</td>
<td>5850</td>
<td>32</td>
<td>0</td>
<td>5</td>
<td>119</td>
<td>17</td>
</tr>
<tr>
<td>Fuel</td>
<td>Object serialization library.</td>
<td>6</td>
<td>131</td>
<td>30</td>
<td>518</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>Glamour</td>
<td>UI framework.</td>
<td>19</td>
<td>465</td>
<td>65</td>
<td>458</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Moose</td>
<td>Software analysis platform.</td>
<td>66</td>
<td>491</td>
<td>120</td>
<td>1091</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>PetriParser2</td>
<td>Parser combinator framework.</td>
<td>14</td>
<td>319</td>
<td>78</td>
<td>1499</td>
<td>349</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Pillar</td>
<td>Document processing platform.</td>
<td>32</td>
<td>354</td>
<td>127</td>
<td>3179</td>
<td>136</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Polymath</td>
<td>Advanced maths library.</td>
<td>54</td>
<td>299</td>
<td>91</td>
<td>767</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>PostgreSQL Parser.</td>
<td>4</td>
<td>130</td>
<td>11</td>
<td>130</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RenoirSt</td>
<td>DSL to generate CSS.</td>
<td>4</td>
<td>105</td>
<td>42</td>
<td>157</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Seaside</td>
<td>Web application framework.</td>
<td>49</td>
<td>837</td>
<td>134</td>
<td>806</td>
<td>44</td>
<td>35</td>
<td>17</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>System</td>
<td>Low-level system packages.</td>
<td>40</td>
<td>260</td>
<td>46</td>
<td>553</td>
<td>11</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Telescope</td>
<td>Visualisation framework.</td>
<td>6</td>
<td>173</td>
<td>21</td>
<td>87</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Zinc</td>
<td>HTTP library.</td>
<td>9</td>
<td>184</td>
<td>43</td>
<td>413</td>
<td>12</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

TABLE I: Characterization of projects under analysis.
primitive on line 6 (in the closure argument of detect:ifNone:) is not executed.

We consider this case to be a false positive, because if the developer had instead written self fail, we would not have classified this test as rotten. It would be easy to identify this pattern in the rotten tests using a static analysis.

b) Missed skip: Several test methods contain guards to stop their execution early under certain conditions. Such a pattern is useful when reusing test suites with different fixures, either to avoid a particular configuration, or to avoid running a test that is not appropriate to the current environment. Listing 9 shows such a case, where the expression thisContext method hasSourceCode guards execution of the test.

c) Context-dependent assertion: The missed skip category is a special case of tests that contain context-dependent assertions. Such tests contain conditionals with different assertions in the different branches. The boolean expressions in these conditionals are either checking that some property of the environment, or, in the case of a helper, checking some condition on the helper’s parameters. Listing 10 shows a context-dependent call to one (assertSerializationIdentityOf:) helper or another (assertSerializationEqualityOf:), depending on the platform on which the test is being executed.

While code like this is not bad per se, identifying such context-dependent tests still provides the developer with important feedback. Indeed, Conditional Logic and Large Fixture are two test smells [17, 34] that can have a detrimental effect on test effectiveness and maintenance. One resolution for this problem is to split the conditional test into two separate tests.

d) Fully rotten tests: This last category describes tests that do not execute one or many assertions and do not fall into any of the three previous categories. Fully rotten tests are caused by actual bugs or logic errors in the test. An example of such a logic error that we have 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 3 is an example of such a rotten test.

C. Experiment Results

Of the 17 projects we analysed, 9 of them contained rotten tests in one or more of the 4 categories. Across the 19,905 tests in these projects, we found 124 missed fail rotten tests, 23 missed skip rotten tests, 134 context dependent rotten tests and 25 fully rotten tests. All in all, 294 rotten tests were found (remember, a test can belong to more than one category).

D. Detailed Project Analysis

In this section, we discuss the results for each project, and some of the more interesting rotten tests.

a) Compiler: this is the default compiler used in Pharo distributions. One rotten test is due to a guard clause creating a context-dependent assertion. The four fully rotten tests concern a feature that is worth examining in some detail.
In Smalltalk, it is normally impossible to use a non-Boolean object in a Boolean expression, even though the implementation uses a Boolean interface. This is because some common boolean methods (such as ifTrue:, and and:) are compiled to optimized bytecodes that raise an exception when they are evaluated for non-booleans. Pharo solves this problem by catching the exception and rewriting the bytecodes using a de-optimization that actually sends the message to the receiver object.

The four rotten tests concern the validation of this feature. A bug in the bytecode-rewrite process induced an early return from the test methods. The return appears before the assertion is executed, and this leads to the 4 tests passing without executing any assertion. Listing 11 shows the source code of one of these methods, both in its original form, and as it is dynamically rewritten. This case shows that it can be extremely difficult to see that an assertion is not executed.

```smalltalk
MustBeBooleanTests » testAnd (original)
| myBooleanObject |
myBooleanObject := MyBooleanObject new.
self deny: (myBooleanObject and: [true])

MustBeBooleanTests » testAnd (rewritten)
| myBooleanObject |
myBooleanObject := MyBooleanObject new.
^ (myBooleanObject) and: [1 halt]
```

Listing 11: a rotten test in Compiler in its original form, and as it is decompiled after bytecode rewriting.

b) **Calypso**: this is Pharo’s new code browser. It allows developers to navigate through, view, and edit packages, classes, and methods. *DrTest* found 88 missed fail tests which contained conditionally-executed sends of TestAssertion-assert: description: with false as the first argument, similar to Listing 8.

c) **Collections**: these packages provide the core data structures of Pharo [7]. *DrTest* found 5 missed skip rotten tests. Tests in Collections are written in a generic way in a test superclass. However, some tests should not be executed on all test subclasses. To skip them on some subclasses, a guard clause is used to check the class in which the test is executed. If the guard is true, the test returns self, thus terminating the test. These 5 missed skip rotten tests are due to these guarded returns, which should be replaced by self.

*DrTest* identified 119 context-dependent rotten tests. Some of those tests could easily be re-written to avoid executing assertions in conditionals. Finally, *DrTest* identified 17 fully rotten tests, all of which contain a bad usage of the Stream API. These methods test the ability of a collection to be serialized on a stream. They do this by (1) creating a stream, (2) writing a collection on the stream and (3) reading what was written on the stream during step 2, and comparing it to the expected result. However, because of a misuse of the Stream API, the assertion inside the comparison loop is never executed. An example of such rotten test is shown in Listing 12.

d) **Fuel**: this is the official library for serializing objects onto the disk. It contains 5 context-dependent rotten tests with assertions that depend on the platform on which the tests are executed. One such was illustrated in Listing 10.

e) **Moose**: this package provides tools and libraries to analyse data and software. It contains one missed-fail rotten test and one fully-rotten test; the latter is shown in Listing 13. Evaluating the argument of the assert:, (self packageP5FullReferee namespaceScope), raises an error; the should:raise: method then catches that error. Thus, assert: is never invoked. The use of assert: in this test is wrong: it is never executed, and even if it were, it would not get a boolean as argument. It is sufficient to write self should: [self packageP5FullReferee namespaceScope] raise: Error.

```smalltalk
FAMIXSelfLoopScopeTest » testFamixPackageNameScope
self should: [self assert: (self packageP5FullReferee namespaceScope)] raise: Error.
```

Listing 13: a rotten test in Moose. An error is raised during the evaluation of the argument of assert:, so assert: is not executed.

f) **PetitParser2**: a parser-combinator framework. It contains one fully-rotten test, shown in Listing 14. This test is defined in the superclass of the class in which it is rotten; it uses inheritance between test classes to drive the execution of the tests. The test iterates over the subclasses of the test class, but one class, PP2SmalltalkParserTests, has no subclass. Thus, the assert: description: in the loop is never executed.

g) **Pillar**: this is a markup syntax and toolkit for writing and generating documentation, books, and slides. One fully-rotten test was found in this project, and is shown in Listing 15. This test is defined in class PRDocumentWriterTest; it is rotten in just one of this class’s subclasses, PRTextWriterTest. This is because, in the context of PRTextWriterTest, self figureBegin is always empty, and so the assertion on line 7 is never executed.

h) **Seaside**: this is a framework for building server-side web applications. *DrTest* found 35 missed fail rotten tests, and 17 missed skip rotten tests. Finally, *DrTest* identified one test, shown in Listing 16, as fully rotten. It turns out that it is in fact a misclassified missed skip rotten test. Looking at the implementation of useCompileUseNewCompiler: during:
PP2SmalltalkGrammarTests>>testCompleteness
    "This test asserts that all subclasses override all test methods."
    | subclasses |
    subclasses := self class allSubclasses.
    ...
    subclasses do: [:subclass |
        self subclass testSelectors do: [:selector |
            self assert: (selector = #testCompleteness or: [ subclass selectors includes: selector ])
        ]
        description: subclass printString , ' does not test ', selector printString ]

Listing 14: Rotten test in PetitParser2. The class in which this method is rotten has no subclass.

PRDocumentWriterTest>>testFigureWithoutLabelAndWithoutCaptionExport
    | item result |
    item := PRFigure new
    reference: 'file://picture.png';
    yourself.
    result := self write: item.
    self figureBegin ifNotEmpty: [ self deny: (result includesSubstring: self figureBegin) ]

Listing 15: Rotten test in Pillar. In PRTextWriterTest the expression self figureBegin is always empty.

shown in Listing 17, we see that the argument aBlock will not be executed if self supportsSwitchToNewCompiler answers false. This explains why the deny and assert: in testCompileByteArrayWithCacheSource are not executed, and the test is classified as fully-rotten by DrTest.

i) System: this project contains the tests for the System packages of Pharo. DrTest detected one missed skip rotten test and 9 context-dependent rotten tests. Those are due to early returns and context-dependent assertions needed to implement these tests generically in a superclass.

WAPharoFileLibraryTest>>useCompileUseNewCompiler:
    aBoolean during: aBlock
    | oldValue |
    self supportsSwitchToNewCompiler
    ifFalse: [ ^ self ].
    "... rest of the helper..."

Listing 17: Helper with guard clause taking a block as parameter.

VI. IMPLEMENTATION

The implementation of DrTest has two key aspects: the detection of helper methods and the tallying of assertion primitive and helper method call site executions. Helper methods are detected through static analysis. Our analysis does not detect helper methods that might be defined in classes outside the test case class hierarchy; this limitation is discussed in Section VII-B.

For the dynamic analysis, the detection of invoked methods is done by instrumenting assertion primitive call sites in the test methods and helper methods. The following subsections provide more details.

A. Detecting Helper Methods

DrTest detects helper methods through static analysis. We first collect all test methods between the analysed test class and the root of the test case hierarchy. Then, we use an AST interpreter to abstractly execute each of those test methods and calculate the transitive closure of message-sends to self and super. The abstract interpreter visits the method AST. Every time it encounters a self or super message-send, it recursively visits the AST for the corresponding method. Finally, we propagate the fact that a helper method invokes an assertion primitive. This results in a list of helper methods that directly or indirectly call assertion primitives.

B. Detecting Assertion Primitive Execution

Assertion primitive execution is detected through bytecode instrumentation. We instrument every assertion primitive call site and helper call-site to set a flag when it is executed. We use the Reflectivity library [16, 40] to do this instrumentation, by adding annotations on the AST nodes that represent the interesting call-sites.

C. The Case of Traits

In Pharo, traits enable developers to reuse tests across several sub-hierarchies; the Collection tests are a good example [18, 19]. The situation with traits is the same as with helper methods: a trait method may show up as being rotten only for certain uses of the trait. This can happen because trait composition and inheritance can change the test fixture. A test method may also be defined in a trait [18], and reused by several classes.

Since traits are implemented by flattening methods in the using class, identifying test and helper methods coming from a trait does not involve any change in our implementation.
However, we have decided to extend our tool to show not only the class where each method is installed, but also its origin. In this way, DrTest shows better information to developers, enabling them to make better decisions.

VII. DISCUSSION AND FUTURE WORK

A. Early Returns in Failing Tests

We decided to ignore skipped tests in our analysis because, by definition, a skipped test will never be executed, and thus neither will its assertions. Skipped tests are reported by the testing framework’s user interface as a separate category, and are not interpreted as passing tests.

Another important point is that skips implemented using a return statement (rather than a send of skip) may occur not only at the beginning of a test, but also in the middle, after some assertions have executed. In such a case, the test will be reported as passing. We consider such a test as rotten, because, depending on the context, it may execute different sets of assertions. Such rotten tests should be refactored into two separate tests, one for each condition.

B. Location and Characterisation of Helper Methods

A limitation of our approach is that our static analysis discovers only those helper methods that are in the hierarchy of the analysed test case. If the helper methods are located in a utility class, they will not be detected.

Although 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 II measures the degree to which this occurs in the projects that we analysed. We computed the number of levels of the class hierarchy between a call to a helper and its definition. 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, etc.

Across all the projects, we found a minimum distance of 0 (all projects using helpers define some of them directly in the test class that uses them), and a maximum distance of 4. We see that helpers are most often used in the class in which they are defined (distance 0).

Identifying helper methods in other hierarchies is difficult, for several reasons. First, Pharo is a dynamically-typed language, like Python, Ruby and Javascript, and inferring the class of non-self sends statically is a complex task [39]. Second, programmers may use reflection; this also renders static analysis ineffective [9]. An alternative to static analysis is to dynamically trace the full system (as opposed to just the test cases), and then to determine statically for each new method executed whether or not it contains a call to an assertion primitive. We leave this solution as future work.

VIII. RELATED WORK

Software testing is an active area of research; researchers have looked at improving the quality of tests, but we are not aware of any prior work that identifies rotten green tests.

A limitation of our approach is that our static analysis discovers only those helper methods that are in the hierarchy of the analysed test case. If the helper methods are located in a utility class, they will not be detected.

Although 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 II measures the degree to which this occurs in the projects that we analysed. We computed the number of levels of the class hierarchy between a call to a helper and its definition. 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, etc.

Across all the projects, we found a minimum distance of 0 (all projects using helpers define some of them directly in the test class that uses them), and a maximum distance of 4. We see that helpers are most often used in the class in which they are defined (distance 0).

Identifying helper methods in other hierarchies is difficult, for several reasons. First, Pharo is a dynamically-typed language, like Python, Ruby and Javascript, and inferring the class of non-self sends statically is a complex task [39]. Second, programmers may use reflection; this also renders static analysis ineffective [9]. An alternative to static analysis is to dynamically trace the full system (as opposed to just the test cases), and then to determine statically for each new method executed whether or not it contains a call to an assertion primitive. We leave this solution as future work.

TABLE II: For each project under analysis, the number of helper calls for the five distances of helper definition observed.

<table>
<thead>
<tr>
<th>Project</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler</td>
<td>58</td>
<td>26</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Aconcagua</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Busy</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Calypso</td>
<td>50</td>
<td>49</td>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Collections</td>
<td>368</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Fuel</td>
<td>13</td>
<td>271</td>
<td>12</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Glamour</td>
<td>157</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Moose</td>
<td>79</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PointParser2</td>
<td>3024</td>
<td>1362</td>
<td>45</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Pillar</td>
<td>1561</td>
<td>518</td>
<td>203</td>
<td>45</td>
<td>3</td>
</tr>
<tr>
<td>Polymath</td>
<td>25</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>3</td>
<td>130</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RemorSt</td>
<td>42</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Seaside</td>
<td>218</td>
<td>127</td>
<td>16</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>System</td>
<td>37</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Telescope</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Zinc</td>
<td>78</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Several researchers have used mutation testing to improve branch coverage [26]. Tillmann et al. [41] 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. [2] present a bacteriological approach to mutation testing. Baudry et al. [3] also worked on improving 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 test adequacy criterion are sufficient to perform diagnosis. This assumption is verified neither by specific experiments nor by intuitive considerations.

c) **Repairing broken tests**: Daniel et al. [14] 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 tests are not broken tests because they are green.

d) **Test smells**: Several works 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. [17] present a list of “bad test smells” and their associated cures, but do not mention rotten green tests. Van Rompaey et al. [35, 36] propose a heuristic, metric-based, approach to identify some test smells (The General Fixture and Eager Tests). They present TestQ, a tool based on static analysis to detect test smells [11]. Bavota et al. [4] present an empirical analysis to assess the presence of test smells and their impact on software maintenance. Reichhart et al. [34] 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 of the proposed smells are advanced. 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 [34]. In this paper we describe how such a smell can be detected automatically. More recently, Bowes et al. [10] 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.

e) **Smoke Tests**: The term “smoke test” is used in various ways by different authors. Waletzky et al. [44] say that the terms Smoke Test and Build Verification Test are sometimes used interchangeably, but prefers to treat Smoke Tests as the subset of Build Verification Tests that are extremely fast to run, and are the prelude to more thorough testing. Other authors use the term “smoke test” to include tests that make assertions. For example, Memon and Xie [27] discuss generating thousands of tests that contain sequences of simulated GUI events, and using various test oracles to check that the state of the GUI is as expected — which they still call smoke tests.

f) **Test effectiveness**: A large body of research has been carried out to assess the effectiveness of tests at detecting faults. To mention a few, Mockus et al. [30] 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 [25] 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. [22] 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.

g) **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. [6] propose to use code coverage for test selection, to maximise the test surface. Blondeau et al. [8] analyse the problem of test selection surfaces in an industrial context.

h) **System robustness**: For test input dependability, using a different approach, Poulding and Feldt [33] increase unit test quality: they automatically generate new invalid and atypical inputs for an 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 [13] and system robustness, Shahrokni et al [38] present RobusTest, a framework to generate automatically tests for timing issues.

i) **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 [21]. When several unit tests in this coverage lattice fail, the tool guides the developer to the test invoking the smallest set of methods.

**IX. Conclusion**

We have identified the existence of rotten green tests, that is, tests that pass and contain assertions, but for which all assertions are not executed. Such tests are worse than no tests at all, because they give developers false confidence in the system under tests. We described an algorithm that identifies rotten green tests, based on a combination of static and dynamic analysis. We presented a tool, DrTest, which implements the proposed approach. DrTest distinguishes rotten green tests from smoke tests (which also execute no assertions, but do so by design). We report on the rotten tests found in 17 large open-source projects based on an analysis containing a total of more than 19900 tests.
REFERENCES


