Rotten Green Tests: A First Analysis

Julien Delplanque
Université de Lille, CRIStAL, CNRS, UMR 9189,
RMoD Team, Inria Lille Nord Europe
Lille, France
julien.delplanque@inria.fr

Stéphane Ducasse
RMoD Team, Inria Lille Nord Europe
Lille, France
stephane.ducasse@inria.fr

Andrew P. Black
Portland State University
Oregon, USA &
RMoD Team, Inria Nord Europe
Lille, France
apblack@pdx.edu

Guillermo Polito
CNRS, Université de Lille, CRIStAL, UMR 9189,
RMoD Team, Inria Lille Nord Europe
Lille, France
guillermo.polito@inria.fr

Abstract

Unit tests are a tenant of agile programming methodologies, and are widely used to improve code quality and prevent code regression. A passing (green) test is usually taken as a robust sign that the code under test is valid. However, we have noticed that some green tests contain assertions that are never executed; these tests pass not because they assert properties that are true, but because they assert nothing at all. We call such tests *Rotten Green Tests*.

Rotten Green Tests represent a worst case: they report that the code under test is valid, but in fact do nothing to test that validity, beyond checking that the code does not crash. We describe an approach to identify rotten green tests by combining simple static and dynamic analyses. Our approach takes into account test helper methods, inherited helpers, and trait compositions, and has been implemented in a tool called *DrTest*. We have applied *DrTest* to several test suites in Pharo 7.0, and identified many rotten tests, including some that have been "sleeping" in Pharo for at least 5 years.

Keywords rotten green tests, unit tests, testing

1 Introduction

Agile methodologies such as Extreme Programming [4–6] promote Unit Testing [22] as a key tenant 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 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 provide assurance that the software is working as expected.

Our concern in this work is with tests that were intended by their designer to execute some assertions, but do not actually do so — we call them *rotten green tests*. 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, a rotten green test guarantees nothing: it is worse than having no test at all!

Our approach is based on a combination of static analysis and dynamic monitoring of method execution. We identify whether or not a test is rotten, even in the presence of helper methods and trait compositions. A limitation of our current implementation is that a test with multiple assertions will be considered as rotten only when *none* of the assertions is executed. Future work will ensure that all the assertions are executed.

The contributions of the paper are:

- the recognition of rotten "green" unit tests that contain assertions that are not executed, and which therefore give the developers false confidence (Section 2);
- a simple combination of static and dynamic analyses that identifies such rotten green tests, (Section 3); and
- a report on our experience applying this approach to several large systems (Section 4).

2 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.

2.1 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 [8, 21]. The following trivial example shows an *SUnit* test that checks that a set should not contain the same object twice.

```
SetTest >> testAddTheSameElementTwiceResultOneOccurrence
  | s |
```

RMOD Inria Technical Report, Jun - 2018,
2018.

```
"Fixture"
s := Set new.
s add: 1.

"Stimulus"
s add: 1.

"Assertions"
self assert: s size equals: 1.
self assert: (s includes: 1).
```

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".

2.2 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, the test ran without raising an unexpected exception [25]. Here is an example of such a test.

```
SetTest >> testSetAddSmokeTest
  | s |
  "Fixture"
  s := Set new.
  s add: 1.

  "Stimulus"
  s add: 1
```

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.

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.

```
TPrintOnSequencedTest >> testPrintOnDelimiter
  | aStream result allElementsAsString |
  result := ''.
  aStream := ReadWriteStream on: result.
  self nonEmpty printOn: aStream delimiter: ', '.
  allElementsAsString := (result findBetweenSubstrings: ', ' ).
  allElementsAsString withIndexDo: [:el :i |
    self assert: el equals: ((self nonEmpty at:i)asString)]
```

Figure 1. A rotten green test

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 which nevertheless does not *execute* any assertions. How can this happen? Let's look at a real example, taken from Pharo issue 7478, and shown in Figure 1.

At first glance, this looks like a fine test of the collection printOn:delimiter: method. The first three lines of the test method create a *fixture* — in this case they set up aStream. Then comes the *stimulus*: the collection self nonEmpty is sent the printOn: aStream delimiter: ',' message, causing it (we hope) to write its elements to aStream separated by commas. The remainder of the test is intended to make *assertions*. The code looks as though it is parsing the contents of aStream and asserting that the elements it finds written there are the same as those in the original collection.

This test is green, so we know that everything is OK, right? Wrong! The programmer who wrote this test misunderstood the way that streams work. The string result can *never* be modified by writing to aStream; indeed, the very 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 run green.

We believe that such a test is worse than no test at all. First, the rotten test assures us of no 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 might reveal that the printOn:delimiter: method is not being exercised, and the developers would at least be aware that their testing is inadequate. Instead, with a rotten test in place coverage will tell us that the printOn:delimiter: method is being *executed* — and we will be unaware that it is not actually being *tested.*

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 Figure 1 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.

3 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. Can't we identify smoke tests as those tests that contain no assertions? No, because many tests make assertions indirectly through the use of helper methods (as explained in Section 3.1); tests that use assertions in helper methods are not smoke tests, even though they contain no assertions directly.

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

- an *assertion primitive* is a method of the unit-testing framework that performs the actual check. In Pharo, assert: and assert:description: are the only assertion primitives.
- a *test method* is a method identified as such by the unit-testing framework. In Pharo, test methods are zero-argument methods defined in a subclass of TestCase 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. In Pharo, SUnit provides helper methods like assert:shouldRaise:; in addition, developers frequently write their own application-specific test helper methods, as we will discuss in Section 3.1.

In Figure 2, testABC is a test method, and helper and secondHelper are helper methods, because helper invokes secondHelper, and secondHelper invokes an assertion primitive.

3.1 Helper methods

It is common practice for developers to factor-out assertions into helper methods. This affects our analysis in two ways: we need to know which messages invoke helper methods when we identify smoke tests, and we must take into account the possibility that helper methods, as well as tests, might be rotten.

What is a helper method? In our context, it is a method that makes an assertion, either using another helper method, or by using one of the assertion primitives. As a simple example, a test suite for approximate numerical methods, might define and use the helper method assert:isRoughly:within:

```
RottenTest >> testABC
   "Test method"
   false ifTrue: [self helper]

RottenTest >> helper
   "Indirect helper"
   self secondHelper

RottenTest >> secondHelper
  "Direct helper"
  self assert: x
```

Figure 2. A rotten test that uses a helper method.

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

For a more realistic example, we turn to the Pillar editing platform [11]. 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 creation of a PRAnchor and its verification. Here is its definition and a sample of its use:

```
PRDocumentWriterTest >> assertWriting: anItem  includesText: aString
  | result |
  result := self write: anItem.
  self assert: result includesSubstring: aString

PRHTMLWriterTest >> testAnchor
  | item |
  item := PRAnchor new name: 'foo'.
  self assertWriting: item includesText: 'id="foo"'
```

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 helper method may depend on the context (*e.g.,* the test fixture, and the arguments to the helper method), a helper might work fine in one context, but be rotten in another. So, when detecting the rotten helper, we need to record the specific test that exposes the rot.

In Pharo, traits enable developers to reuse tests across several sub-hierarchies; the Collection tests are a good example [15, 16]. The situation with traits is the same as with helper methods: a trait method may show up as being rotten only for a certain trait use. This can happen because trait composition and inheritance can change the test fixture.

3.2 Classifying Tests

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

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

Rotten tests. A test passes, and contains assertions (either directly, or indirectly), but no assertion is executed: the test, or the helper method, is rotten.

Smoke tests. 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.

3.3 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 identifying all the methods of the unit test framework that make assertions directly. This set depends only on the test framework.

Step 2: **identification of helper methods.** To build the set of helper methods, we start with a set *S* containing all the methods in the test class that are not test methods but which self-send one of the assertion primitives. ("All the methods" includes methods introduced through inheritance or trait use.) We identify such methods using a simple static analysis of self- and super-sends. We then add to *S* any method in the test class that is not a test case but which self- or super-sends one of the methods in *S*. We repeat this step until no new methods are added to *S*; the set *S* now contains all the helper methods.

Step 3: **test execution.** We execute each test method (including inherited test methods) one at at time, while monitoring the following pieces of information:

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

b. whether or not one of the *helper methods* is executed, and

c. whether or not one of the *assertion primitives* is executed.

Because we are looking for rotten *green* tests, we consider only passing tests.

Step 4: **classification.** We combine the test execution information from Step 3 with the static information collected in Steps 1 and 2. Table 1 shows how this information is used to classify test methods. We discuss the classification in Section 3.4.

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. A test method may also be defined in a trait [15], and reused by several classes. In general, the test fixture, and thus the meaning of the test, will be different in each place in which it is 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 we can understand the scenario in which the helper fails to make an assertion.

Our current dynamic analysis has a coarse granularity: we monitor the execution of primitive assertions for the complete run of the test method under analysis. This means that we cannot attribute an executed assertion to a specific call site. So, if one primitive assertion in the test method was executed, the test will not be classified as rotten, even though there might be another primitive assertion that was *not* executed. We discuss this shortcoming further in Section 6.

3.4 Classification Discussion

We now discuss the reasoning behind the classification in Table 1. In the first four rows, the test contains (statically) sends of both assertion primitives and helper methods. If the test is good, we will see both helper and assertion executed, as in row 1. If no helper, or no assertion primitive, or neither, is executed, then the test is rotten.

In rows 5–8, there is no send to a helper method in the test, but there is a send of an assertion primitive. If an assertion is executed (rows 5 and 6), the test is good; if no assertion is executed (rows 7 and 8), we can say that the test is rotten. On rows 5 and 7, a helper method is executed even though there is no send of a helper message in the test! How can this happen? One possibility is that the test constructs the selector of the helper method dynamically, and then uses perform: to send it. Another possibility is that the test invokes a helper method with a message send that is not a self- or super-send, either because the message is actually sent to an object outside its class hierarchy, or because the receiver of the message is not the pseudo-variable self. We flag these entries with "dynamic helper invocation"; test like this are unlikely to occur in practice, but if they do, they are worth a second look.

Rows 9–12 are similar to rows 1–4, except that in rows 9–12 all of the assertions are made using helper methods. If both helper and assertion are executed (row 9), the test is good; if either is not executed, something is rotten. In row 11, the helper is executed, but the assertion is not, so it is the helper method that is rotten. In rows 10 and 12, no helper is executed, even though the test relies on helper methods, so we know that the test is rotten. This reasoning applies even in row 10 where, somehow, an assertion is executed even though no assertion message is sent!

In rows 13–16, no helpers and no assertions are used in the test. Such tests look like smoke tests — in which case we would expect that no helpers and no assertions are executed,

	Dynamic Analysis		Static Analysis		
Row №	Helper Executed	Assertion Executed	Test contains helper	Test contains assertion	Classification
1	✓	✓	✓	✓	✓Good test
2	✗	✓	✓	✓	✗Rotten test
3	✓	✗	✓	✓	✗Rotten test & rotten helper
4	✗	✗	✓	✓	✗Rotten test
5	✓	✓	✗	✓	✓Good test (dynamic helper invocation)
6	✗	✓	✗	✓	✓Good test
7	✓	✗	✗	✓	✗Rotten test & rotten helper (dynamic helper invocation)
8	✗	✗	✗	✓	✗Rotten test
9	✓	✓	✓	✗	✓Good test
10	✗	✓	✓	✗	✗Rotten test (dynamic assertion invocation)
11	✓	✗	✓	✗	✗Rotten helper
12	✗	✗	✓	✗	✗Rotten test
13	✓	✓	✗	✗	✓Good test (dynamic assertion & helper)
14	✗	✓	✗	✗	✓Good test (dynamic assertion invocation)
15	✓	✗	✗	✗	✓Good test (dynamic helper invocation)
16	✗	✗	✗	✗	✓Smoke test

Table 1. Classifying tests using the results of our analyses. In all cases, the test passes. "Executed" means that the (helper or assertion) method was executed. "Contains" means that the test method contains (statically) a message send to a helper or to an assertion.

as occurs on row 16. If a helper or assertion *is* executed, it is likely to be because of a dynamic invocation.

Let us see how this classification works on the example in Figure 2. The dynamic analysis will report that neither a helper method nor an assertion primitive is executed. The static analysis will identify secondHelper as a helper method (because it makes a primitive assertion), and helper as a helper method (because it uses secondHelper). Hence, we look for ✗✗✓✗ in Table 1, which we find on row 12. As expected, this tells us that we have found a rotten test.

4 Results

Pharo is an open-source language with a growing community and a large number of mature projects. We have run *DrTest* on eight Pharo subsystems; we found that all subsystems except Iceberg have at least 1 rotten test. The results are shown in Table 2.

Subsystem	Packages	Classes	Test classes	Tests	Rotten tests
Calypso	58	705	128	2671	4
Collections	16	224	59	5858	7
Glamour	19	463	65	449	3
Iceberg	16	565	44	555	0
Opal Compiler	7	227	49	854	15
Pillar	33	358	112	3188	1
System	48	330	44	552	1
Zinc	9	184	43	412	3

Table 2. Rotten tests in Pharo subsytems.

```
ClyCompositeScopeTests>>#testEmptySubscopesAreForbidden
  [ClyCompositeScope on: #().
     self assert: false description: 'empty subscopes should be forbid-
den'] ifError: [].
```

Figure 3. Rotten test in Calypso. The assertion is not reached on purpose, this is a false positive.

Calypso is the new system browser for Pharo. It allows developers to view and edit packages, classes, and methods. *DrTest* found four rotten tests; two of them contained conditionally-executed sends of TestAsserter » assert:description: with false as the first argument, as shown in Figure 3. These two tests are false positives: the developer intends that the assertion will not be executed if the code under test behaves correctly, that is, if it signals an exception. This test would be clearer if it simply asserted that ClyCompositeScope on: #() should raise an error.

The two other rotten tests in Calypso contain helper methods that are never executed. These tests are tagged with the <expectedFailure> pragma which indicates that they are expected to fail by SUnit. Again, those are false positives. Andrew ▶*I don't see why, but then, I can't find the tests.*◀

Collections are the basic data structures of Pharo [8]. The collections tests contain seven rotten tests. Two of them are tests that contain a guard clause checking which of the collection classes is being tested. This guard clause is present because the test is implemented in a superclass common to multiple test cases, but the test should be run only on certain subclasses. One rotten test contains a guard clause

```
TPrintTest >> testPrintElementsOn
  | aStream result allElementsAsString tmp |
  result:=''.
  aStream:= ReadWriteStream on: result.
  tmp:= OrderedCollection new.
  self nonEmpty do: [:each | tmp add: each asString].

  self nonEmpty printElementsOn: aStream .
  allElementsAsString:=(result findBetweenSubstrings: ' ' ).
  1 to: allElementsAsString size do: [:i |
    self assert: (tmp occurrencesOf:(allElementsAsString at:i))
      = (allElementsAsString  occurrencesOf:(allElementsAsString at:i)).
  ].
```

Figure 4. A Rotten test of Pharo Collections. The stream API is used incorrectly.

that deliberately skips the test when the Dictionary subclass under test does not support nil keys.

The remaining four rotten tests are true positives that use the Stream API incorrectly. These methods test the serialization of a collection on a stream, using the following process: (1) create a stream, (2) write a collection on the stream and (3) compare what was written on the stream to the original collection. As discussed in Section 2.3, the problem here is that the comparison should be with the contents of the stream, aStream contents, and not with the variable result, which remains bound to the empty string. Because of this, the assertion inside the loop is never executed. Figure 4 illustrates another such rotten test.

Glamour provides a high-level API for creating user interfaces. It has three rotten tests. After an investigation, we realised that two of them are expected failures. Andrew ▶*So what? If the tests fail, then they can't be rotten, so I assume that they pass. Are they rotten?*◀ The last one is because a block containing an assertion is never executed Andrew ▶*Is this a true positive? Lets say so*◀.

Iceberg is a tool to manage Git project within Pharo. It was found to contain no rotten tests.

OpalCompiler is the default compiler used in Pharo images. It has 15 rotten tests; 10 of them are due to a guard clause that was incorrect. As shown in Figure 5, the comment above the guard clause specifies that the test should not be run unless the method's source code is available. But the implementation of the condition in the guard clause ensures that the test is *skipped* if the source code is available.

The other five rotten tests of the OpalCompiler were intended to identify a bug related to dynamic bytecode rewriting of boolean expressions. In Smalltalk, it is normally impossible to make a non-Boolean object act as a Boolean in a condition, even if the object implements the Boolean interface. The reason is that common Boolean message like #ifTrue: and #and: are compiled to optimized bytecode that raises an exception when they are sent to non-Booleans. However, Pharo dynamically catches this exception and rewrites the bytecode to a de-optimized version. This allows one to use a non-Boolean receiver as a Boolean (provided that it implements the methods of Boolean).

```
OCContextTempMappingTest >>
  testAccessingArgOfOuterBlockFromAnotherDeepBlock
  | actual |
   "Check the source code availability to do not fail on images without sources"
  thisContext method hasSourceCode ifTrue: [ ^ self ].

  actual := [:outerArg |
    outerArg asString.
    [ :innerArg | innerArg asString.
      thisContext tempNamed: #outerArg ]
        value: #innerValue.
    ] value: #outerValue.

  self assert: actual equals: #outerValue
```

Figure 5. Rotten test in OpalCompiler. The guard clause implements the negation of what is described in the comment.

```
MustBeBooleanTests >> testAnd
  | myBooleanObject |

  myBooleanObject := MyBooleanObject new.
  self deny: (myBooleanObject and: [true])

MustBeBooleanTests >> testAnd (rewritten)
  | myBooleanObject |

  myBooleanObject := MyBooleanObject new.
  ^ (myBooleanObject) and: [ 1 halt ]
```

Figure 6. Rotten test in OpalCompiler: as written (top) as decompiled after rewriting (bottom).

The five rotten OpalCompiler tests are intended to validate this feature. A bug in the bytecode rewrite process introduced an early return into the test methods. The return appears before the assertion can be executed. Thus, it leads to five tests passing but not executing any assertion. Figure 6 shows the source code of one of these five methods, both as it appears, and as it is dynamically rewritten. These rotten tests teach us that it can be extremely difficult to see whether or not an assertion is executed.

Pillar is a markup syntax and associated tools for writing and generating documentation, books, and slides. This project contains 1 rotten test which is tagged as an expected failure. Andrew ▶*again, what has expected failure to do with things?*◀

System packages are also provided with test suites. These test suites contain one rotten test, which is tagged as an expected failure. Andrew ▶*again, what has expected failure to do with things?*◀

Zinc is a library for interacting with the network. It contains three rotten tests. One of them is a test that sends #skip, a method of TestAsserter whose specification says “Don’t run this test, and don’t mark it as failure”. Thus, we can be confident that the assertions in this test are skipped on purpose.

Another Zinc test is guarded by a check that a certain class exists in the system; if it does not, the test returns. By default, the class does not exist, so the test returns and no assertion is executed.

The last rotten test of Zinc performs a network HTTP operation. If the result is a redirection, the test returns immediately. The URL requested by the test does lead to a redirection, so the test returns and no assertion is executed.

5 Implementation

The implementation of *DrTest* has two points of interest: the detection of helper methods and the detection of method execution. Helper methods are detected through static analysis: we traverse the class hierarchy starting from the subclass of TestCase that interests us, and classify as helper methods all non-test methods that make a direct or indirect send of an assertion primitive. For the dynamic analysis, the detection of method execution is done by instrumenting the assertion primitives and helper methods. We replace those methods by method *spies* that record that they were called, and then forward execution to the original method.

5.1 The Algorithm in Outline

To identify the rotten tests of a test class, *DrTest* starts by determining which methods are helpers (Section 5.2) and marking them with spy objects containing a flag called (Section 5.3). Then, *DrTest* executes each test case by:

1. reseting all spies so that called = false,
2. running the test, which will mark executed method by setting called = true, and
3. classifying the outcome of the test using the marked spies according to Table 1.

The code that classifies a test outcome is as follows.

- **Table Rows 1–4.**

```
(containsAssertionPrimitive and: [
  containsHelper and: [
    (assertionPrimitiveExecuted and: [
      helperExecuted ]) not ] ])
        ifTrue: [ self addRottenTest: compiledMethod ].
```

- **Table Rows 7–8.**

```
(containsAssertionPrimitive and: [
  (containsHelper not) and: [
    assertionPrimitiveExecuted not ] ])
      ifTrue: [ self addRottenTest: compiledMethod ].
```

- **Table Row 11.**

```
(containsAssertionPrimitive not and: [
  containsHelper and: [
    assertionPrimitiveExecuted not and: [
      helperExecuted ] ] ])
        ifTrue: [ self addRottenHelper: compiledMethod ].
```

- **Table Rows 10 and 12.**

```
(containsAssertionPrimitive not and: [
  containsHelper and: [
    helperExecuted not ] ])
      ifTrue: [ self addRottenTest: compiledMethod ].
```

5.2 Detecting Helper Methods

DrTest detects helper methods through static analysis. The analysis first collects all the test methods Andrew ▶*do we mean the* non-test *methods?*◀ in the test class under analysis, and all of its superclasses up to TestAsserter (which is the superclass of TestCase). All of these methods may be relevant to the execution of the test class under analysis. We then use an AST visitor to visit the entire AST of each of these methods; every time the visitation encounters a self-send, it notes the method corresponding to the self-send. This gives us an enumeration of the relation “executes directly via a self-send”. We then compute the transitive closure of this relation, and note the methods that directly or indirectly self-send an assertion primitive. The noted methods are the helper methods.

5.3 Detecting Method Execution

The method spy mechanism uses the method wrapper virtual machine (VM) hook described by Martinez Peck et al. [19], which works as follows. When a message is sent to an object, the VM looks in the method dictionary of the receiver’s class for an entry keyed by the method selector. In most cases, this entry will be a compiled method, that is, an instance of the class CompiledMethod, in which case the VM executes the method with the arguments from the message. However, when the object found in the method dictionary is *not* a compiled method, the VM instead sends the message run:with:in: to the found object, giving it a chance to execute some behaviour. This hook is commonly used by profilers and other dynamic analysis tools.

In our case, the object in the method dictionary is an instance of MethodTracer, a class that implements a run:with:in: method that contains a called flag, which is initialized to false, and a reference to the original method. When a MethodTracer is sent a run:with:in: it first “marks” itself by setting the called flag to true, and then forwards execution to the original compiled method. Here is the relevant code from MethodTracer:

```
MethodTracer >> run: aSelector  with: someArguments  in: aReceiver
  self mark.
  ^ aReceiver withArgs: someArguments executeMethod: method
```

Distinct instances of MethodTracer are used to spy on the assertion primitives and on the helper methods. This allows us to keep a separate record of whether or not assertion primitives and helper methods have been executed.

6 Discussion and Future Work

As currently implemented, *DrTest* has two significant limitations.

6.1 Tests containing multiple assertions

When a test method uses multiple assertion primitives or helper methods, our approach does not currently distinguish between the case in which just one assertion primitive or helper methods is executed, and the case where they are all executed. In both cases, we report that the test is good.

For example, the test in Figure 1 is rotten, and is detected as such. However, if the developer had added

```
self assert: (self nonEmpty isNotEmpty) description: 'test fixture broken'
```

at the top of the test, it would *not* be detected as rotten, even though this assertion does nothing to solve the problem. Indeed, because it is based on method-level monitoring, the current implementation cannot distinguish between assertions made at different call sites. This can result in false-negatives, that is, not all rotten tests will be detected. It will not cause false-positives: a test that is identified as rotten will always be rotten.

We cannot yet evaluate the number of false negatives due to multiple assertions, because doing so would require that we enhance our solution to eliminate them.

For the future, we can see several ways of dealing test that contain with multiple assertions. One possibility is to create, from each test method that contains multiple assertions, multiple tests, each with a single assertion, and then to apply our process to these new methods. Another possibility is to record, not just when *an* assertion is executed, but *which* send of the assertion message is being executed. These enhancements are the subject of ongoing work.

6.2 Location of Helper Methods

Another 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, then they will not be detected. We can estimate how frequently this problem arises because some instances will manifest themselves as a test method *executing* an assertion even though it *does not contain* any assertions or helper methods — the cases classified as involving a dynamic invocation in Table 1. (Tests that contain self-sends that execute helper methods, *and* sends of helper messages to a utility object, will not be detected.) Our preliminary results indicate that these cases do not occur frequently.

7 Related Work

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

Mutation testing is one of the earliest approaches used to improve test quality and robustness [13]. Mutation testing generates “mutant” programs from the program under test, and then selects tests that differentiate the mutants from the original program. Tests that detect few or no mutants are candidates for removal.

Several researchers have used mutation testing to improve branch coverage [18]. Tillmann and Schulte [23] use symbolic execution to find inputs for parameterized unit tests that achieve high code coverage. They turn existing unit tests into parameterized unit tests and generate entirely new parameterized unit tests that describe the behaviour of an existing implementation.

Baudry et al. [2] present a bacteriological approach to mutation testing.

Other approaches focused on other attributes of test quality. Baudry et al. [3] addresses improving the value of tests for diagnosis. They propose a new attribute called the Dynamic Basic Block, to assist in the task of locating faults in a program. 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.

Often, unit test frameworks present failed unit tests in an arbitrary order, but developers want to focus on the most specific ones first. Gaelli et al. [17] propose a partial order of unit tests corresponding to the containment hierarchy of their sets of covered method signatures. When several unit tests fail in the same hierarchy, the tool can guide the developer to the test involving the smallest number of methods.

Other work focuses on the selection of the tests to be run. For example, when a change is made to some software, it is desirable to rerun those tests that are most likely to be invalidated by the change. Beszedes et al. [7] propose to use code coverage for test selection testing to maximize the test surface. Blondeau et al. [9] analyse the problem of test selection surfaces in an industrial context. Verhaeghe et al. [24] examined the practices of developers when writing code, and the way they execute (or choose not to execute) tests. SmartTest [12, 24] is a tool that automatically selects the tests to be run.

Deursen et al. [14] present a list of “bad test smells” and their associated cures. They do not mention rotten green tests as a smell.

The term “smoke test” is used in various ways by different authors. Waletzky [25] says 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 [20] 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.

The implementation of *DrTest* uses a form of method wrapper to count executions of helper methods and assertions. Brant et al. [10] discuss many techniques for implementing wrappers. Our approach relies on a Pharo VM hook that was not available to Brant et al., who instead subclassed compiled method. Martinez Peck et al. [19] describe the VM hook used by *DrTest*; they use it to implement *Ghost* proxies.

8 Conclusion

We have identified the existence of Rotten Green Tests, that is, tests that pass and contain assertions, but whose 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 have described an algorithm that identifies Rotten Green Tests, based on a combination of static and dynamic analysis, and a tool, *DrTest*, that 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 tests found in eight large open-source projects.

DrTest is publicly available on github and can be loaded from github://juliendelplanque/RottenTestsFinder/src.

Acknowledgements

This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council, CPER Nord-Pas de Calais/FEDER DATA Advanced data science and technologies 2015-2020. Andrew Black was able to spend time at Inria because of a sabbatical granted by Portland State University.

References

[1] David Astels. 2003. *Test-Driven Development — A Practical Guide.* Prentice Hall.

[2] Benoit Baudry, Franck Fleurey, Jean-Marc Jézéquel, and Yves Le Traon. 2005. Automatic Test Case Optimization: A Bacteriologic Algorithm. *IEEE Software* 22, 2 (2005), 76–82.

[3] Benoit Baudry, Franck Fleurey, and Yves Le Traon. 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, 82–91. https://doi.org/10.1145/1134285.1134299

[4] Kent Beck. [n. d.]. Manifesto for Agile Software Development. ([n. d.]). http://agilemanifesto.org.

[5] Kent Beck. 2002. *Test Driven Development: By Example.* Addison-Wesley Longman.

[6] Kent Beck and Cynthia Andres. 2004. *Extreme Programming Explained: Embrace Change (2Nd Edition).* Addison-Wesley Professional.

[7] A. Beszedes, T. Gergely, L. Schrettner, J. Jasz, L. Lango, and T. Gyimothy. 2012. Code Coverage-based Regression Test Selection and Prioritization in WebKit. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on.* 46–55. https://doi.org/10.1109/ICSM.2012.6405252

[8] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. 2009. *Pharo by Example.* Square Bracket Associates, Kehrsatz, Switzerland. 333 pages. http://rmod.inria.fr/archives/books/Blac09a-PBE1-2013-07-29.pdf

[9] Vincent Blondeau, Anne Etien, Nicolas Anquetil, Sylvain Cresson, Pascal Croisy, and Stéphane Ducasse. 2016. Test Case Selection in Industry: An Analysis of Issues Related to Static Approaches. *Software Quality Journal* (2016), 1–35. https://doi.org/10.1007/s11219-016-9328-4

[10] John Brant, Brian Foote, Ralph Johnson, and Don Roberts. 1998. Wrappers to the Rescue. In *Proceedings European Conference on Object Oriented Programming (ECOOP'98) (LNCS)*, Vol. 1445. Springer-Verlag, 396–417.

[11] Damien Cassou, Stéphane Ducasse, Luc Fabresse, Johan Fabry, and Sven Van Caekenberghe. 2015. *Enterprise Pharo: a Web Perspective.* Square Bracket Associates.

[12] Serge Demeyer, Benoît Verhaeghe, Anne Etien, Nicolas Anquetil, and Stéphane Ducasse. 2018. Evaluating the Efficiency of Continuous Testing during Test-Driven Development. In *Proceedings VST 2018 (2nd IEEE International Workshop on Validation, Analysis and Evolution of Software Tests).* 1 – 5. https://hal.inria.fr/hal-01717343

[13] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer* 11, 4 (April 1978), 34–41. https://doi.org/10.1109/C-M.1978.218136

[14] Arie van Deursen, Leon Moonen, Alex van den Bergh, and Gerard Kok. 2001. Refactoring Test Code. In *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes (XP2001)*, M. Marchesi (Ed.). University of Cagliari, 92–95.

[15] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black. 2006. Traits: A Mechanism for fine-grained Reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 28, 2 (March 2006), 331–388. https://doi.org/10.1145/1119479.1119483

[16] Stéphane Ducasse, Damien Pollet, Alexandre Bergel, and Damien Cassou. 2009. Reusing and Composing Tests with Traits. In *TOOLS'09: Proceedings of the 47th International Conference on Objects, Models, Components, Patterns.* Zurich, Switzerland, 252–271. http://rmod.inria.fr/archives/papers/Duca09a-Tools2009-TraitTests.pdf

[17] Markus Gaelli, Michele Lanza, Oscar Nierstrasz, and Roel Wuyts. 2004. Ordering Broken Unit Tests for Focused Debugging. In *20th International Conference on Software Maintenance (ICSM 2004).* 114–123. https://doi.org/10.1109/ICSM.2004.1357796

[18] R. Lingampally, A. Gupta, and P. Jalote. 2007. A Multipurpose Code Coverage Tool for Java. In *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on.* 261b–261b. https://doi.org/10.1109/HICSS.2007.24

[19] Mariano Martinez Peck, Noury Bouraqadi, Luc Fabresse, Marcus Denker, and Camille Teruel. 2015. Ghost: A Uniform and General-Purpose Proxy Implementation. *Journal of Object Technology* 98 (2015), 339–359. Issue 3. https://doi.org/10.1016/j.scico.2014.05.015

[20] Atif M. Memon and Qing Xie. 2004. Empirical Evaluation of the Fault-Detection Effectiveness of Smoke Regression Test Cases for GUI-Based Software. In *IEEE Intl Conf. on Software Maintenance.* 8–17.

[21] Gerard Meszaros. 2007. *XUnit Test Patterns – Refactoring Test Code.* Addison Wesley.

[22] G. Meszaros, S.M. Smith, and J. Andrea. 2003. The Test Automation Manifesto. In *Proceedings of the Third XP and Second Agile Universe Conference.* 73–81.

[23] Nikolai Tillmann and Wolfram Schulte. 2005. Parameterized unit tests. In *ESEC/SIGSOFT FSE.* 253–262. ftp://ftp.research.microsoft.com/pub/

tr/TR-2005-64.pdf

[24] Benoit Verhaeghe, Nicolas Anquetil, Stéphane Ducasse, and Vincent Blondeau. 2017. Usage of Tests in an Open-Source Community. In *Proceedings of the 12th Edition of the International Workshop on Smalltalk Technologies (IWST '17)*. ACM, New York, NY, USA, Article 4, 9 pages. https://doi.org/10.1145/3139903.3139909

[25] James Waletzky. 2012. Smoke Tests vs. BVTs. Crosslake Tech Blog. (Apr 2012). http://www.crosslaketech.com/smoke-vs-bvt/