Rotten Green Tests

Julien Delplanque, Stéphane Ducasse, Guillermo Polito, Andrew Black, Anne Etien

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 tenant 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 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 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 provide assurance that the software is working as expected.

The software engineering research community has developed many techniques to help assess the quality of test cases. Some approaches are based on mutation testing [15, 26, 2, 3]; others on code coverage [30, 6, 41] or on repairing tests [14]. Several researchers [17, 29, 34, 42, 4, 10] have worked on test smells (following Fowler et al.'s Code Smells [20]). Huo et al. [24] present OraclePolish, an approach and tool to identify 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 identify rotten green tests is based on a combination of a simple static analysis and dynamic monitoring of method execution at call-site granularity. We 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 occur in Pharo [7], rotten green tests are not specific to Pharo; we encourage replication of our approach in other languages.

The contributions of the paper 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.

Section II provides a small introduction to Pharo, the language in which the projects that we analysed are written. Section III describes the anatomy of a unit test, and compares rotten tests with smoke tests. Section IV introduces the approach we developed to automatically identify rotten tests. Section V presents the experiment we used validate our approach, and its results. Section VI describes the implementation of *DrTest*. Section VII discusses some aspects of our approach and future work. Section VIII presents related work.

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; field access is like Java's protected, but fields are never visible to other classes in the package. In addition, a class 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.methodN1n2(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 return statement uses the caret construct ^; 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 that checks that a set should not contain the same object twice.

```
1 SetTest » testSetDuplication
2   | s |                          "Local variable definition"
3   s := Set new.                  "Fixture"
4   s add: 1.
5   s add: 1.                      "Stimulus"
6   self assert: s size equals: 1.  "Assertions"
7   self assert: (s includes: 1).
```

Listing 1: the parts of a unit test.

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; and 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.

```
1 SetTest » testSetAddSmokeTest
2   | s |
3   s := Set new.     "Fixture"
4   s add: 1.
5   s add: 1.         "Stimulus"
```

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.

```
1  TPrintOnSequencedTest » testPrintOnDelimiter
2    | aStream result allElementsAsString |
3    result := ''.
4    aStream := ReadWriteStream on: result.
5    self nonEmpty printOn: aStream delimiter: ', '.
6    allElementsAsString := result findBetweenSubstrings: ', '.
7    allElementsAsString withIndexDo: [:el :i |
8      self
9        assert: el
10       equals: ((self nonEmpty at:i) asString) ]
```

Listing 3: a rotten green test.

At first glance, this looks like a fine test of the collection printOn:delimiter: 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 written on it are the same as 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 run 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:delimiter: 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 TestAsserter (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 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. Developers frequently write application-specific test helper methods, as we discuss in Subsection IV-A.
- a *rotten test* is a test that passes, contains assertions (either directly, or indirectly through a helper), but in which at least one assertion is not executed.

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.

```
1  NumericalTests » assert: actual isRoughly: desired within: eps
2    self
3      assert: (actual - desired) abs <= eps
4      description: [ 'actual result ' , actual ,
5        ' 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.

```
1  PRDocumentWriterTest » assertWriting: item includesText: str
2    | result |
3    result := self write: item.
4    self assert: result includesSubstring: str
5
```

```
6 PRHTMLWriterTest » testAnchor
7   | item |
8   item := PRAnchor new name: 'foo'.
9   self assertWriting: item includesText: 'id="foo"'
```

Listing 5: a helper method for html generation

Like a rotten test, a helper method might fail to make an assertion in some or all situations; 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 method), we need to record the specific test that exposes the issue, because a helper might work fine in one context, but be rotten in another.

B. Classifying Tests

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

Good tests. 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 tests. 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 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.

Listing 6 shows that 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. In addition, testABC contains a valid assertion self assert: true. While this assertion is correctly executed, the call to method helper will not be executed, and as a consequence the assertion in secondHelper will not be executed. Thus, this test contains one assertion that is not executed, which makes it rotten by our definition. This example shows that the analysis should distinguish between the different assertion call-sites.

```
1  RottenTest » testABC
2    "Test method"
3    false ifTrue: [self helper]
4    self assert: true.
5  RottenTest » helper
6    "Indirect helper"
7    self secondHelper
8  RottenTest » secondHelper
9    "Direct helper"
10   self assert: x
```

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

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 the call sites to assertion primitives and helpers in the test method and helper methods. That is to say, we set up a mechanism that allows us to determinate if, after the execution of a method, all the call sites of assertion primitives and helpers have 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 during test execution.

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.

							found rotten tests			
Project	Description	#pack.	#classes	#test	#tests classes	#helpers	missed fail	missed skip	context dependent	fully rotten
Compiler	AST model and compiler of Pharo.	6	232	51	859	10	0	0	1	4
Aconcagua	Model representing measures.	2	84	27	661	2	0	0	0	0
Buoy	Various package extensions	12	51	19	185	0	0	0	0	0
Calypso	Pharo IDE.	58	705	157	2692	4	88	0	0	0
Collections	Pharo collection library.	16	222	59	5850	32	0	5	119	17
Fuel	Object serialization library.	6	131	30	518	4	0	0	5	0
Glamour	UI framework.	19	463	65	458	9	0	0	0	0
Moose	Software analysis platform.	66	491	120	1091	6	1	0	0	1
PetitParser2	Parser combinator framework.	14	319	78	1499	349	0	0	0	1
Pillar	Document processing platform.	32	354	127	3179	136	0	0	0	1
Polymath	Advanced maths library.	54	299	91	767	3	0	0	0	0
PostgreSQL	PostgreSQL Parser.	4	130	11	130	2	0	0	0	0
RenoirSt	DSL to generate CSS.	4	103	42	157	4	0	0	0	0
Seaside	Web application framework.	49	837	134	806	44	35	17	0	1
System	Low-level system packages	40	260	46	553	11	0	1	9	0
Telescope	Visualisation framework.	6	173	21	87	0	0	0	0	0
Zinc	HTTP library.	9	184	43	413	12	0	0	0	0

TABLE I: Characterization of projects under analysis.

```
1 RottenTest » testDEF
2   "Test method"
3   self badHelper.
4   self assert: true.
5 RottenTest » badHelper
6   "Indirect helper"
7   false ifTrue: [ self secondHelper ]
8 RottenTest » secondHelper
9   "Direct helper"
10  self assert: x
```

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

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.

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 describes some of the rotten tests that we found.

A. Characterizing the projects under analysis

Pharo is an open-source language 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 Subsection 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 detect:ifNone: 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.

```
1 TSequencedElementAccessTest »
      test0FixtureSequencedElementAccessTest
2 self moreThan4Elements.
```

```
3  self assert: self moreThan4Elements size >= 4.
4  self subCollectionNotIn
5    detect: [:each | (self moreThan4Elements includes: each) not]
6    ifNone: [self assert: false].
7  self elementNotInForElementAccessing.
8  self deny: (self moreThan4Elements includes: self
       elementNotInForElementAccessing).
9  self elementInForElementAccessing.
10 self assert: (self moreThan4Elements includes: self
       elementInForElementAccessing)
```

Listing 8: a missed fail (on line 6).

To make this test fail, the developer used **self** assert: **false** instead of **self** fail; this is a misuse of the testing framework. Our approach classifies this test as rotten because the assertion primitive called from 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 written **self** fail we would not have classified this test as rotten. Identifying this pattern amongst the rotten tests is easily automated 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 fixtures, either to avoid a particular configuration, or to avoid running some test depending on the developer's environment. Listing 9 shows such a case, where the expression **thisContext** method hasSourceCode guards execution of the test.

```
1  OCContextTempMappingTest »
2   testAccessingArgOfOuterBlockFromAnotherDeepBlock
3   | actual |
4   "Check the source code availability to do not
5   fail on images without sources"
6   thisContext method hasSourceCode
7     ifTrue: [ ^ self ].
8   actual := [:outerArg |
9     outerArg asString.
10    [ :innerArg | innerArg asString.
11      thisContext tempNamed: #outerArg ]
12        value: #innerValue.
13  ] value: #outerValue.
14  self assert: actual equals: #outerValue
```

Listing 9: a rotten test in Compiler. The guard clause (lines 6 and 7) does not implement what is described in the comment above: the ifTrue: should be an ifFalse:.

To show the intent to avoid running a test, the testing framework provides the primitives skip and skip:, but the test in Listing 9 does not use them. *DrTest* therefore detects the test as rotten, again because of a misuse of the testing framework. To be explicit and help the code's maintainer, the guarded expression should be **self** skip rather than ^ **self**. Note that the writer of this test got the condition backwards; this illustrates the danger of guarded tests that return rather than skip. It is possible to detect this pattern automatically too, by searching for tests containing a return statement.

*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 properties of the environment are fulfilled, 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.

```
1  FLBasicSerializationTest » testCharacter
2  "Test character serialization. If the code is less than 255 the
       same instance is used. But if it is bigger, new ones are
       created."
3  self assertSerializationIdentityOf: $a.
4  FLPlatform current isSpur
5    ifTrue: [ self assertSerializationIdentityOf: (Character value:
         12345). "Japanese Hiragana 'A' " ]
6    ifFalse: [ self assertSerializationEqualityOf: (Character value:
         12345). "Japanese Hiragana 'A' " ].
7  self assertSerializationEqualityOf: Character allCharacters.
8  self assertSerializationEqualityOf: (Array with: $a with: (
       Character value: 12345)).
```

Listing 10: a rotten test in the Fuel framework; one of the two blocks of the conditional on lines 4–6 is not 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 (a context-dependent assertion). The four fully-rotten tests were intended to identify a bug related to the dynamic bytecode rewriting of boolean expressions.

In Smalltalk, it is normally impossible to make a (non-Boolean) object implementing the Boolean interface act like a Boolean in an expression. The reason is that common boolean methods (*e.g.,* ifTrue:, and:) are compiled to optimized bytecodes that raise an exception when they are evaluated for non-booleans. However, Pharo dynamically catches this exception and rewrites the bytecodes using a de-optimization that actually sends the message to the receiver object.

These rotten tests concern the validation of this feature. A bug in the bytecode-rewrite process induced an early return from these 4 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. What this case shows is that it can be extremely difficult to see that an assertion is not executed.

```
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 TestAsserter»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 super test class. However, some tests can not be executed on all subclasses. To skip them on some test subclasses, a guard clause is used to test the class in which the test is executed. This guard clause, if the condition is true, leads to a return of self to interrupt the execution of the test. These 5 missed skip rotten tests are due to these early returns, which should be replaced by calls to the skip method.

DrTest identified 119 context-dependent rotten tests. Some of those tests could be easily 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.

```
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)) ].
```

Listing 12: a rotten test in Pharo Collections. On line 8, rather than looking for substrings in result, the test should look in aStream contents.

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. An example of such a test can be seen 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 can be seen in Listing 13. When the argument of the assert: (**self** packageP5FullReferee namespaceScope) is executed, it raises an error. Then, the should:raise: method catches the Error. Thus, assert: is never called. 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.

```
FAMIXSelfLoopScopeTest»testFamixPackageNamespaceScope
 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, which is defined in the superclass of the class in which it is rotten. This test uses the class hierarchy and inheritance between test classes to drive the execution of tests. It iterates on the subclasses of the test class, but some classes, such as PP2SmalltalkParserTests, have no subclasses. Thus, the assert:description: in the loop is never executed. Listing 14 shows the rotten test.

g) Pillar: this is a markup syntax and associated tools to write and generate documentation, books, and slides. One fully-rotten test was found in this project. This test is defined in the superclass of the class in which it is rotten. Listing 15 shows the rotten test. PRTextWriterTest is the subclass of PRDocumentWriterTest in which this test is rotten. When the test is executed in the context of PRTextWriterTest class, **self** figureBegin is always empty. The test is thus rotten.

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 1 fully-rotten test (see Listing 16). It turns out that it is in fact a misclassified missed skip rotten test. Looking at the implementation of useCompileUseNewCompiler:during: (Listing 17),

```
PP2SmalltalkGrammarTests»testCompleteness
"This test asserts that all subclasses override all test methods."
| subclasses |
subclasses := self class allSubclasses.
"..."
subclasses do: [ :subclass |
  self class 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.

we can see that the block provided as parameter might not be executed if self supportsSwitchToNewCompiler answers false . This explains why the deny: and assert: in testCompileByteArrayWithCacheSource are not executed, and the rotten 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»
      testCompileByteArrayWithCacheSource
| library data file source expected |
self
  useCompileUseNewCompiler: false
  during: [
    self deny: GRPharoPlatform current useByteArrayLiterals.
    "..."
    [ source := WATestingFiles sourceCodeAt: #demoJpeg.
    expected := 'demoJpeg
^ #(1 2 "...truncated..." 254 255) asByteArray'.
    self assert: source greaseString = expected ]
  ensure: [ library removeFile: file fileName ] ]
```

Listing 16: Rotten test in Seaside.

```
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 methods call site executions. Helper methods are detected through static analysis: we traverse the class hierarchy starting from the test case class that interests us, and select all methods that arrive at an assertion primitive through a chain of self-sends. Our analysis does not support helper methods that might be defined in classes outside the test case class hierarchy (See 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 test case hierarchy. Then, we use an AST interpreter to abstractly execute each of those test methods and perform the transitive closure of self / super message-sends. This abstract interpreter recursively visits the entire method AST. Every time it encounters a message-send, if it is a self-send it recursively visits it, otherwise it is ignored. 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 mark with a flag if they were 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 a certain trait use. 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 for each method not only the class where it is installed but also its origin. In this way, *DrTest* will show better information to developers, so they can 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, then 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).

Providing a general solution to identify helper methods in other hierarchies is difficult. 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; again this 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.

Project	0	1	2	3	4
Compiler	58	26	0	0	0
Aconcagua	4	0	0	0	0
Buoy	0	0	0	0	0
Calypso	50	49	10	0	0
Collections	368	0	0	0	0
Fuel	13	271	12	0	0
Glamour	157	0	0	0	0
Moose	79	0	0	0	0
PetitParser2	3024	1362	45	0	0
Pillar	1561	518	203	45	3
Polymath	25	0	0	0	0
PostgreSQL	3	130	0	0	0
RenoirSt	42	0	0	0	0
Seaside	218	127	16	0	0
System	37	1	0	0	0
Telescope	0	0	0	0	0
Zinc	78	0	0	0	0

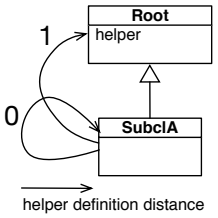

TABLE II: For each project under analysis, the number of helper calls for the five distances of helper definition observed.

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) ***Test analysis****:* Herzig et al. [23] present an approach based on machine learning to detect false test alarms: false test alarms are integration tests that are directly linked to code defects but fail due to external factors such as hardware failing to give access to a test resource. Identifying false test alarms is key since they demand the attention of engineers, and require manual checking to determine that they are false alarms.

Vera Perez et al. [43] present a novel analysis of pseudo-tested methods. These are methods that are covered by tests, yet no test case fails when the method body is removed. This intriguing concept was named in 2016 by Niedermayr et al., who showed that such methods are systematically present, even in well-tested projects with high statement coverage [31]. Pseudo-tested methods are only indirectly related related to rotten tests: one possible cause of a pseudo-tested method is that all the tests of that method are rotten. The existence of pseudo-tested methods shows that it is important to improve tool support for testing.

Huo et al. [24] present an approach based on taint analysis to identify brittle or unused test inputs. They monitor the flow of controlled and uncontrolled inputs along data and control dependencies. Our approach focuses on monitoring assertion execution. From that perspective it is complementary. Mockus et al. [30] present an approach to determine test effectiveness. Their analysis compares test coverage prior to a release with the number of reported failures after that same release.

We agree with Schuler and Zeller that simple coverage is not enough, especially in presence of dynamic dead code [37]. Both approaches identify tests that make no assertions. Their approach is good for improving code coverage by identifying program elements that are not in the dynamic-backward slice of any assertion. Our approach does not look for coverage improvement. We do not look for statements that are not covered by tests. We assure programmers that when a test is green, its assertions have been executed. Our analysis is less general, but also much simpler to implement, which we hope will make it more likely to be adopted.

b) ***Mutation testing****:* Mutation testing is one of the earliest techniques used to improve test quality and robust-

ness [15]. 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 unit tests into parametrized unit tests, and generate entirely new parametrized unit tests that describe the behavior 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] also worked on improving test-for-diagnosis criterion: they propose a new attribute called 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 test smells too. However, none of the existing catalogs mention rotten tests. Deursen et al. [17] present a list of "bad test smells" and their associated cures. They do not mention rotten green tests as a smell. 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 testing principles to capture the essence of testing goals. They identified that 8 principles are not covered by existing test smells and they proposed some metrics to identify such smells.

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 have 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. It 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

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

[2] B. Baudry, F. Fleurey, J.-M. Jézéquel, and Y. L. Traon. Automatic test case optimization: A bacteriologic algorithm. *IEEE Software*, 22(2):76–82, 2005.

[3] B. Baudry, F. Fleurey, and Y. L. Traon. Improving test suites for efficient fault localization. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 82–91, New York, NY, USA, 2006. ACM Press.

[4] G. Bavota, A. Qusef, R. Oliveto, A. D. Lucia, and D. Binkley. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *International Conference on Software Maintenance (ICSM)*, pages 56–65. IEEE, sep 2012.

[5] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change (2Nd Edition)*. Addison-Wesley Professional, 2004.

[6] A. Beszedes, T. Gergely, L. Schrettner, J. Jasz, L. Lango, and T. Gyimothy. Code Coverage-based Regression Test Selection and Prioritization in WebKit. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 46–55, sep 2012.

[7] A. P. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker. *Pharo by Example*. Square Bracket Associates, Kehrsatz, Switzerland, 2009.

[8] V. Blondeau, A. Etien, N. Anquetil, S. Cresson, P. Croisy, and S. Ducasse. Test case selection in industry: An analysis of issues related to static approaches. *Software Quality Journal*, pages 1–35, 2016.

[9] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 241–250, New York, NY, USA, 2011. ACM.

[10] D. Bowes, H. Tracy, J. Petrié, T. Shippey, and B. Turhan. How good are my tests? In *Workshop on Emerging Trends in Software Metrics (WETSoM)*. IEEE/ACM, 2017.

[11] M. Breugelmans and B. Van Rompaey. TestQ: Exploring structural and maintenance characteristics of unit test suites. In *International Workshop on Advanced Software Development Tools and Techniques (WASDeTT)*, 2008.

[12] D. Cassou, S. Ducasse, L. Fabresse, J. Fabry, and S. Van Caekenberghe. *Enterprise Pharo: a Web Perspective*. Square Bracket Associates, 2015.

[13] C. Csallner and Y. Smaragdakis. Jcrasher: an automatic robust tester for java. *Software: Practice and Experience*, 43, 2004.

[14] B. Daniel, D. Dig, T. Gvero, V. Jagannath, J. Jiaa, D. Mitchell, J. Nogiec, S. H. Tan, and D. Marinov. Reassert: A tool for repairing broken unit tests. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1010–1012, New York, NY, USA, 2011. ACM.

[15] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, Apr. 1978.

[16] M. Denker. *Sub-method Structural and Behavioral Reflection*. PhD thesis, University of Bern, May 2008.

[17] A. Deursen, L. Moonen, A. Bergh, and G. Kok. Refactoring test code. In M. Marchesi, editor, *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes (XP2001)*, pages 92–95. University of Cagliari, 2001.

[18] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. P. Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):331–388, Mar. 2006.

[19] S. Ducasse, D. Pollet, A. Bergel, and D. Cassou. Reusing and composing tests with traits. In *TOOLS'09: Proceedings of the 47th International Conference on Objects, Models, Components, Patterns*, pages 252–271, Zurich, Switzerland, June 2009.

[20] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[21] M. Gaelli, M. Lanza, O. Nierstrasz, and R. Wuyts. Ordering broken unit tests for focused debugging. In *20th International Conference on Software Maintenance (ICSM 2004)*, pages 114–123, 2004.

[22] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M. A. Alipour, and D. Marinov. Comparing non-adequate test suites using coverage criteria. In *International Symposium on Software Testing and Analysis*, 2013.

[23] K. Herzig and N. Nagappan. Empirically detecting false test alarms using association rules. In *International Conference on Software Engineering*, 2015.

[24] C. Huo and J. Clause. Improving oracle quality by detecting brittle assertions and unused inputs in tests. In *Foundations on Software Engineering*, 2014.

[25] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *International Conference on Software Engineering*, 2014.

[26] R. Lingampally, A. Gupta, and P. Jalote. A multipurpose code coverage tool for java. In *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*, pages 261b–261b, jan 2007.

[27] A. M. Memon and Q. Xie. Empirical evaluation of the fault-detection effectiveness of smoke regression test cases for gui-based software. In *IEEE International Conference on Software Maintenance*, pages 8–17, 2004.

[28] G. Meszaros. *XUnit Test Patterns – Refactoring Test Code*. Addison Wesley, June 2007.

[29] G. Meszaros, S. Smith, and J. Andrea. The test automation manifesto. In *Proceedings of the Third XP and Second Agile Universe Conference*, pages 73–81, Aug. 2003.

[30] A. Mockus, N. Nagappan, and T. T. Dinh-Trong. Test

coverage and post-verification defects: A multiple case study. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ESEM '09, pages 291–301, Washington, DC, USA, 2009. IEEE Computer Society.

[31] R. Niedermayr, E. Juergens, and S. Wagne. Will my tests tell me if i break this code? In *International Workshop on Continuous Software Evolution and Delivery*, pages 23–29. ACM Press, 2016.

[32] L. S. Pinto, S. Sinha, and A. Orso. Understanding myths and realities of test-suite evolution. In *International Conference on Software Engineering*, 2012.

[33] S. M. Poulding and R. Feldt. Generating controllably invalid and atypical inputs for robustness testing. In *IEEE International Conference on Software Testing, Verification and Validation Workshops*, pages 81–84, 2017.

[34] S. Reichhart, T. Gîrba, and S. Ducasse. Rule-based assessment of test quality. In *Journal of Object Technology, Special Issue. Proceedings of TOOLS Europe 2007*, volume 6/9, pages 231–251, Oct. 2007. Special Issue. Proceedings of TOOLS Europe 2007.

[35] B. V. Rompaey, B. D. Bois, and S. Demeyer. Characterizing the relative significance of a test smell. *icsm*, 0:391–400, 2006.

[36] B. V. Rompaey, B. D. Bois, and S. Demeyer. Improving test code reviews with metrics: a pilot study. Technical report, Lab On Re-Engineering, University Of Antwerp, 2006.

[37] D. Schuler and A. Zeller. Checked coverage: an indicator for oracle quality. *Software testing, verification and reliability*, 23:531–551, 2013.

[38] A. Shahrokni and R. Feldt. Robustest: Towards a framework for automated testing of robustness in software. In *International Conference on Advances in System Testing and Validation LifeCycle*, 2011.

[39] S. A. Spoon and O. Shivers. Demand-driven type inference with subgoal pruning: Trading precision for scalability. In *Proceedings of ECOOP'04*, pages 51–74, 2004.

[40] É. Tanter, J. Noyé, D. Caromel, and P. Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In *Proceedings of OOPSLA '03, ACM SIGPLAN Notices*, pages 27–46, nov 2003.

[41] N. Tillmann and W. Schulte. Parameterized unit tests. In *ESEC/SIGSOFT FSE*, pages 253–262, 2005.

[42] B. Van Rompaey, B. Du Bois, S. Demeyer, and M. Rieger. On the detection of test smells: A metrics-based approach for general fixture and eager test. *Transactions on Software Engineering*, 33(12):800–817, 2007.

[43] O. Vera-Perez, B. Danglot, M. Monperrus, and B. Baudry. A comprehensive study of pseudo-tested methods. *CoRR*, abs/1807.05030, 2018.

[44] J. Waletzky. Smoke tests vs. BVTs. Crosslake Tech Blog, Apr 2012. http://www.crosslaketech.com/smoke-vs-bvt/.