ROTTEN GREEN TESTS

J. Delplanque, S. Ducasse, G. Polito, A. P. Black and A. Etien
V. Aranega, M. Martinez, C. Fuhrman

Univ. Lille, CNRS, Centrale Lille, Inria, UMR 9189 - CRIStAL
Dept of Computer Science, Portland State University, Oregon, USA
WHO AM I?

➤ Professor at the University of Lille from Sept 2020

➤ Member of the RMod team joint with Inria Lille Nord Europe and University of Lille.

➤ Topics:

➤ Architecture Remodularisation

➤ Software Transformations

➤ Database Evolution

➤ UI Migration

➤ API Update

➤ Tests selection

Maintenance and evolution of legacy software systems
WHAT IS A ROTTEN GREEN TEST?
ANATOMY OF A TEST

SetTest » testSetAdd

| s |
s := Set new.
s add: 1.
s add: 1.
self assert: s size equals: 1.
self assert: (s includes: 1)

In Pharo
s add: 1. <=> s.add(1);

class SetTest {
    method testSetAdd {
        def s = Set.new()
        s.add(1)
        s.add(1)
        self.assertEquals(s.size(), 1)
        self.assert(s.includes(1))
    }
}
SetTest » testSetAddSmokeTest

| s |

s := Set new.
s add: 1.
s add: 1

➤ No assertion
➤ Not a rotten green test
A ROTTEN GREEN TEST IS

- A test *passing (green)*
- A test that contains at least one *assertion*
- One or more assertions is *not* executed when test runs
A LITTLE SKETCH OF A ROTTEN GREEN TEST

class RottenTest {
    method testABC {
        if (false) then {self.assert(x)}
    }
}

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) ]
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) ]

Not executed!
The programmer believed that the object on which the stream is working is “magically” mutated on stream growth.
@Test
public void testLoggerContainsLogEntry(){
    Logger logger = new Logger();
    logger.log("log1");
    logger.log("log2");
    for (LogEntry logEntry : logger.getLogEntries()) {
        assertTrue(logger.containsLogEntry(logEntry));
    }
}

Sometime returns empty collection
ROTTEN GREEN TEST WRITERS

➤ Rotten green tests are NOT intentional

➤ We say: this is *not* the programmer’s fault

➤ Instead: it is the fault of testing tools that *do not report* them
WHY ARE ROTTEN GREEN TESTS BAD?

➤ Give a false sense of security
➤ Can easily pass unnoticed
➤ Not reported by testing frameworks prior to DrTest in Pharo and RTJ in Java.
MAINLY CAUSED BY

➤ Conditional code not executing a branch
➤ Iterating over an empty collection
HOW TO IDENTIFY THEM?

**ROTTEN GREEN TEST IS...**

- A test *passing (green)*
- A test that contains at least one *assertion*
- One or more assertions is *not* executed when test runs
class RottenTest {
    method testABC {
        if (false) then {self.helper()}
    }

    method helper {
        self.secondHelper()
    }

    method secondHelper {
        self.assert(x)
    }
}
class RottenTest {
    method testABC {
        if (false) then {self.helper()}
    }

    method helper {
        self.secondHelper()
    }

    method secondHelper {
        self.assert(x)
    }
}

Not executed!
Not executed!
ABOUT THE NEED FOR CALL SITE ANALYSIS

class RottenTest {
    method testDEF {
        self.badHelper()
        self.assert(true)
    }

    method badHelper {
        if (false) then {
            self.secondHelper()
        }
    }

    method secondHelper {
        self.assert(x)
    }
}
class RottenTest {
    method testDEF {
        self.badHelper()
        self.assert(true) // Executed!
    }

    method badHelper {
        if (false) then {
            self.secondHelper() // Not executed!
        }
    }

    method secondHelper {
        self.assert(x) // Not executed!
    }
}
IDENTIFYING ROTTEN GREEN TESTS

➤ We use both

➤ *Static analysis*, to identify helpers and inherited methods

➤ *Dynamic analysis*, to identify *call* sites that are not executed
IDENTIFYING ROTTEN GREEN TESTS

➤ Static Analysis
  ➤ Identify “testing primitives” (assert:, deny:…)
  ➤ Identify helper methods (abstract interpreter)
➤ Dynamic Analysis through instrumentation
  ➤ Instrument the call-sites of the “test primitives”
➤ Run the test suite
  ➤ Record green tests whose test primitives are not executed
➤ Generate Report
class RottenTest {
    method testDEF {
        self.badHelper()
        self.assert(true)
    }

    method badHelper {
        if (false) then {
            self.secondHelper()
        }
    }
}

method secondHelper {
    self.assert(x)
}
}
class RottenTest {
    method testDEF {
        self.badHelper()
        self.assert(true)
    }

    method badHelper {
        if (false) then {
            self.secondHelper()
        }
    }

    method secondHelper {
        self.assert(x)
    }
}
class RottenTest {
    method testDEF {
        self.badHelper()
        self.assert(true)
    }

    method badHelper {
        if (false) then {
            self.secondHelper()
        }
    }

    method secondHelper {
        self.assert(x)
    }
}
class RottenTest {
    method testDEF {
        self.badHelper()
        self.assert(true)
    }

    method badHelper {
        if (false) then {
            self.secondHelper()
        }
    }

    method secondHelper {
        self.assert(x)
    }
}
IDENTIFICATION IN JAVA

➤ JUnit 4.X testing framework

➤ Identification of assertion primitives
  ➤ static method invocations whose name start with the keyword `assert` and the class of the invocation’s target is `org.junit.Assert`.

➤ Identification of helper methods
  ➤ Method with body containing an assertion or an invocation to a helper method.

➤ Call-site instrumentation

➤ Test execution
  ➤ Comparison between lines of code executed and loc not.
IDENTIFICATION IN PYTHON

➤ Unitest and pytest framework.

➤ Identification and instrumentation of call-sites
  ➤ In the AST of the test, each method call is instrumented with a metabehavior that spreads itself at run time and identify assertion primitive

➤ Test execution

➤ Identification of assertion primitives (depending from the used framework)

➤ Dynamical identification of helper methods
CATEGORISATION
Missing fail: test passes false to assert:, instead of using fail. Such assertions should be executed only if something goes wrong.

```java
public void testHasProtectedConstructor () {
    try {
        ConstructorAccess<HasProtectedConstructor> access =
            ConstructorAccess.get( HasProtectedConstructor.class);
        HasProtectedConstructor newInstance = access.newInstance();
        assertEquals("cow",
            newInstance.getMoo());
    }
    catch (Throwable t) {
        System.out.println("Unexpected exception happened: " + t);
        assertEquals(false);
    }
}

A missed fail from project Reflectasm.
```
SOLVING MISSING FAIL

➤ Test passes false to `assert`:

➤ Check on a case by case basis

➤ Use `fail` to really indicate that the assertion should not be executed
ROTTEN TESTS CATEGORISATION

➤ Missed skip: test contains guards to stop its execution early (under certain conditions), instead of using skip

```java
@Test public void testNormalizedKeyReadWriter() {

   ...

    TypeComparator<T> comp1 = getComparator(true);
    if(!comp1.supportsSerializationWithKeyNormalization()){
        return;
    }

    ...

    assertTrue(comp1.compareToReference(comp2) == 0);
    ...
}
```

A skip rotten test in Apache-Flink
SOLVING MISSED SKIP

➤ Test contains guards to stop its execution early (under certain conditions), instead of using `skip`
➤ Easy to fix: use `self skip`
➤ Runner can then report correctly
**Context-dependent logic:** complex logic with different assertions in different branches — some may be rotten too

```java
@Test public void testCoGroupLambda() {
    CoGroupFunction<Tuple2<...» f = (i1, i2, o) --> {};
    TypeInformation<??> ti = TypeExtractor.getCoGroupReturnTypes(f, ...);
    if (!(ti instanceof MissingTypeInfo)) {
        assertTrue(ti.isTupleType());
        assertEquals(2, ti.getArity());
        ...
    }
}
```
SOLVING CONTEXT DEPENDENT LOGIC

➤ Complex logic with different assertions in different branches — some may be rotten too

➤ How to fix:
  ➤ create a separate test for each branch
  ➤ use `self skip` to execute the test only when it applies
**ROTTEN TESTS CATEGORISATION**

➤ **Fully rotten:** other tests that do not execute one or more assertions

```python
class TestDot11Decoder(unittest.TestCase):
    def setUp(self):
        self.WEPKey=None #Unknown #
        other inits
    def test_04_Dot11WEPData(self):
        if not self.WEPKey:
            return
        self.assertEqual(str(self.in3.__class__), "impacket.dot11.Dot11WEPData")
        # Test if wep data "get_packet" is correct
        wepdata=b'\x6e\xdf\x93\x36\x39\x5a\x39\x66\x6b\x96\xda\xb6\x11\nx22\xfd\xf0\xd4\x0d\x6a\xb8\xb1\xe6\xe2\xe1f\xe25\x7d\x64\x1a\x07\xda5\x86\xda2\nx19\x34\xb5\xf7\x8a\x62\x33\x59\x6e\xe89\x01\x73\x50\x12\xbb\xde\x17'
        self.assertEqual(self.in3.get_packet(),wepdata)
```

*A rotten green test with the unittest framework*
FULLY ROTTEN TESTS

➤ Other tests that do not execute one or more assertions

```ruby
testFamixPackageNamespace
  self
  should: [ self assert: self packReferee ]
  raise: Error

➤ packageP5FullReferee *did* raise Error, so assert: is never invoked!
```
Other tests that do not execute one or more assertions

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

Compiled method bytecode was dynamically rewritten by the compiler as:

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

*Oops the deny: disappeared!*
SOLVING FULLY ROTTEN

➤ Case-by-case analysis and repair
EVALUATION
# CASE STUDIES (CHECK THE PAPER)

- 19,905 tests analysed on mature projects
- 294 rotten (25 fully rotten)

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

➤ RQ1: Do rotten green tests exist in languages other than Pharo?

➤ RQ2: Do we find the same categories of rotten green tests as in the original paper?

➤ RQ3: Do rotten green tests hide bugs in the program or only in the tests?
PROJECT SELECTION

➤ Selection of around 100 projects per language (Java and Python)

➤ Projects available on GitHub

➤ Popular but also representative of the language or the community.

➤ At least a push after 2019/1/1

➤ Computation of a score and taking the 100 first projects
  
  score = mergePullRequestsCount/10 + closedIssuesCount/10 + forkCount + starsCount.
EXTRA CRITERIA

➤ For Pharo —> 113 projects
  ➤ Relax date of push after 2017/1/1
  ➤ Pharo version >= 2015
  ➤ Continuous integration
  ➤ Tests can be run locally and all are passing.

➤ For Java —> 67 projects
  ➤ JUnit 4.x as testing framework
  ➤ Maven as a dependency manager system

➤ For Python —> 100 projects
  ➤ Pytest or unittest as testing framework
  ➤ Existence has a setup.py file
## RQ1 (PRESENCE) AND RQ2 (CATEGORISATION)

<table>
<thead>
<tr>
<th>Language</th>
<th>Missed fail</th>
<th>Missed skip</th>
<th>Context Dependent</th>
<th>Fully Rotten</th>
<th>Projects</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Tests</td>
<td>Proj</td>
<td>Tests</td>
<td>Proj</td>
<td>Tests</td>
</tr>
<tr>
<td>Pharo</td>
<td>180</td>
<td>6</td>
<td>57</td>
<td>8</td>
<td>321</td>
</tr>
<tr>
<td>Java</td>
<td>13</td>
<td>5</td>
<td>51</td>
<td>7</td>
<td>254</td>
</tr>
<tr>
<td>Python</td>
<td>212</td>
<td>12</td>
<td>71</td>
<td>17</td>
<td>831</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
RQ3 (BUGS)

➤ In Pharo

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

➤ Compiled method bytecode was dynamically rewritten by the compiler as:

MustBeBooleanTests » testAnd (rewritten)
  | myBooleanObject |
  myBooleanObject := MyBooleanObject new.
  ^ (myBooleanObject) and: [true]
In Python in the junja2 project

This test ensures that if a template contains more than one extends, a dedicated TemplateError is raised.

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

Using else branch or pytest.raises context manager

Manually modifying the code turn the test red!
CONCLUSION

➤ Rotten Green Tests exist.

➤ Every Unit-testing framework should report them

➤ DrTests the new Pharo Unit framework reports them

➤ Replicated the experiment on Java and Python

➤ Found many rotten green tests in these two languages

➤ Categories are the same

➤ Bugs hidden by rotten green tests where found in Pharo and Python