Test Smells Catalog

Release 2.0

EASY LAB - Universidade Federal de Alagoas (UFAL)

Aug 02, 2023

TEST SMELL CATEGORIES

Welcome!

Search for a test smell using the **search** or the **sidebar** — works with test smells descriptions too — if you already know what to look for. If not, you may browse by category in the left panel. Download options (PDF and EPUB) are available in the *Read the Docs* menu (bottom of left panel). There is a **lot** to see!

The catalog provides a visualization of the data set compiled in a Multivocal Literature Review published here. Initially, the catalog unifies information from 127 formal and informal sources. As the catalog is open-source, any community member can submit a contribution as a pull request and help improve it.

Every listed test smell contains a name, AKA (when available), definition, code example (likely extracted from the listed references), and the listed references, which of course provide further information. We expect the catalog to be a helpful resource for software testing community members to better understand test smells. Furthermore, it may help avoid proposing new names (AKA) to existing test smells in an overlap.

Tip: Please help us keep this Catalog updated and **contribute with your own test smell findings**.

Send in your helpful comments or ideas to easy@ic.ufal.br or contribute directly by clicking *Edit on GitHub* in the top right corner of this page.

If you are not familiar with directly editing, give us a new issue! Click here and create a new GitHub issue

CHAPTER

ONE

CONTENTS

1.1 Code related

1.1.1 Code duplication

1.1.1.1 Army Of Clones

Definition:

- Different tests perform and implement similar actions, leading to duplicated pieces of test code

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- On the Maintenance of System User Interactive Tests
- Smells in System User Interactive Tests

1.1.1.2 Assertion Chorus

Definition:

- where a series of assertions repetitively perform a long winded routine to test something.

Also Known As:

- Missing custom assertion method

References:

Quality attributes

- - Code Example
- - Cause and Effect

- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.1.1.3 Copy-Paste

Definition:

- Not only for test development, copy-paste bugs are also very common in product code. Just do a copy-paste without enough modification carelessly

Code Example:

- No code examples yet…

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Development Patterns and Anti-Patterns - Medium

1.1.1.4 Cut-And-Paste Code Reuse

Definition:

- Cut and Paste is a powerful tool for writing code fast but it results in many copies of the same code each of which must be maintained in parallel.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- xUnit test patterns: Refactoring test code

1.1.1.5 Data-Ja Vu

Definition:

- where some data that could be made immutable and loaded once is read for every instance of a test

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.1.1.6 Duplicate Alt Branches

Definition:

- Different alt constructs contain duplicate branches

Code Example:

```
testcase tc_example_TestCase1() runs on ExampleComponent {
    timer t_guard ;
    // . . .
    t_guard.start(10.0) ;
    alt{
        [] pt.receive(a_MessageOne){
        pt.send(a_MessageTwo);
        }
        [] any port.receive {
        set.verdict(fail);
        stop;
        }
        [] t_guard.timeout{
        set.verdict(fail);
        stop;
    }
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- An approach to quality engineering of TTCN-3 test specifications
- Pattern-based Smell Detection in TTCN-3 Test Suites
- Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites

1.1.1.7 Duplicate Assert

Definition:

- This smell occurs when a test method tests for the same condition multiple times within the same test method. If the test method needs to test the same condition using different values, a new test method should be created. As a best practice, the name of the test method should be an indication of the test being performed. Possible situations that would give rise to this smell include (1) developers grouping multiple conditions to test a single method, (2) developers performing debugging activities, and (3) an accidental copy-paste of code

Code Example:

```
@Test
public void testXmlSanitizer() {
    boolean valid = XmlSanitizer.isValid("Fritzbox");
    assertEquals("Fritzbox is valid", true, valid);
    System.out.println("Pure ASCII test - passed");

    valid = XmlSanitizer.isValid("Fritz Box");
    assertEquals("Spaces are valid", true, valid);
    System.out.println("Spaces test - passed");

    valid = XmlSanitizer.isValid("Frützbüx");
    assertEquals("Frützbüx is invalid", false, valid);
    System.out.println("No ASCII test - passed");

    valid = XmlSanitizer.isValid("Fritz!box");
    assertEquals("Exclamation mark is valid", true, valid);
    System.out.println("Exclamation mark test - passed");

    valid = XmlSanitizer.isValid("Fritz.box");
    assertEquals("Exclamation mark is valid", true, valid);
    System.out.println("Dot test - passed");

    valid = XmlSanitizer.isValid("Fritz-box");
    assertEquals("Minus is valid", true, valid);
    System.out.println("Minus test - passed");

    valid = XmlSanitizer.isValid("Fritz-box");
    assertEquals("Minus is valid", true, valid);
    System.out.println("Minus test - passed");
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect

- - Frequency
- - Refactoring

- An Exploratory Study on the Refactoring of Unit Test Files in Android Applications
- Analyzing Test Smells Refactoring from a Developers Perspective
- Automatic Identification of High-Impact Bug Report by Product and Test Code Quality
- Automatic generation of smell-free unit tests
- Characterizing High-Quality Test Methods: A First Empirical Study
- Characterizing High-Quality Test Methods: A First Empirical Study
- Handling Test Smells in Python: Results from a Mixed-Method Study
- Investigating Test Smells in JavaScript Test Code
- On the Distribution of "Simple Stupid Bugs" in Unit Test Files: An Exploratory Study
- On the diffusion of test smells and their relationship with test code quality of Java projects
- On the distribution of test smells in open source Android applications: an exploratory study
- On the influence of Test Smells on Test Coverage
- On the test smells detection: an empirical study on the jnose test accuracy
- On the use of test smells for prediction of flaky tests
- PyNose: A Test Smell Detector For Python
- Pytest-Smell: a smell detection tool for Python unit tests
- RAIDE: a tool for Assertion Roulette and Duplicate Assert identification and refactoring
- Refactoring Test Smells With JUnit 5: Why Should Developers Keep Up-to-Date?
- Software Unit Test Smells
- Test Smell Detection Tools: A Systematic Mapping Study
- The secret life of test smells-an empirical study on test smell evolution and maintenance
- Understanding practitioners' strategies to handle test smells: a multi-method study
- What the Smell? An Empirical Investigation on the Distribution and Severity of Test Smells in Open Source Android Applications
- tsDetect: an open source test smells detection tool

1.1.1.8 Duplicate Code

Definition:

- For test automation, it's especially easy to find yourself with the same code over and over. Steps such as logging in and navigating through common areas of the application are naturally a part of most scenarios. However, repeating this same logic multiple times in various places is a code smell. Should there be a change in the application within this flow, you'll have to hunt down all the places where this logic is duplicated within your test code and update each one. This takes too much development time and also poses the risk of you missing a spot, therefore introducing a bug in your test code.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Writing good gherkin

1.1.1.9 Duplicate Component Definition

Definition:

- Two or more components declare identical variables, constants, timers or ports.

Code Example:

```
type component c1 {
    var integer i ;
    const integer id := 1;
    timer t;
    port ExamplePort p1;
}

type component c2 {
    const integer id := 2;
    timer t;
    port ExamplePort p2;
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- An approach to quality engineering of TTCN-3 test specifications
- Pattern-based Smell Detection in TTCN-3 Test Suites
- Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites

1.1.1.10 Duplicate In-Line Templates

Definition:

- Two or more similar or identical in-line templates.

Code Example:

```
module DuplicateInlineTemplates (
    type record ExampleRecordType (
        boolean exampleField1,
        integer exampleField2,
        charstring exampleField3
}
type port ExamplePort message (
    out ExampleRecordType;

}
type component ExampleComponent (
    port ExamplePort pt;
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- An approach to quality engineering of TTCN-3 test specifications
- Pattern-based Smell Detection in TTCN-3 Test Suites
- Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites

1.1.1.11 Duplicate Local Variable/Constant/Timer

Definition:

- The same local variable, constant or timer is defined in two or more functions, test cases or altsteps running on the same component.

Code Example:

```
type component c {
    port ExamplePort p;
}
testcase tc1() runs on c {
    timer t;
    p.send("foo1");
    t.start (10.0);
    alt {
```

(continues on next page)

(continued from previous page)

```
        [] p.receive("bari") {
        // do something
        }
        [] any port.receive{
        // error handling
        }
        [] t.timeout {
        // error handling
        }
    }
}
testcase tc2() runs on c {
    timer t;
    p.send("foo2");
    t.start (20.0);
    alt {
        [] p.receive("bar2") {
        // do something
        }
        [] any port.receive {
        // error handling
        }
        [] t.timeout {
        // error handling
        }
    }
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- An approach to quality engineering of TTCN-3 test specifications
- Pattern-based Smell Detection in TTCN-3 Test Suites
- Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites

1.1.1.12 Duplicate Statements

Definition:

- A duplicate sequence of statements in the statement block of one or multiple behavioral entities (functions, test cases and altsteps).

Code Example:

```
function f_sendMessages(in float p_duration) runs on ExampleComponent {
    timer t;
    t.start(p_duration);
    t.timeout;
    pt.send("first timeout");
    t.start( p_duration );
    t.timeout;
    pt.send("second timeout " );
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- An approach to quality engineering of TTCN-3 test specifications
- Pattern-based Smell Detection in TTCN-3 Test Suites
- Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites

1.1.1.13 Duplicate Template Fields

Definition:

- The fields of two or more templates are identical or very similar

Code Example:

```
template MyRecordType t1 := {
field1 := "foo",
field2 := 1,
field3 := true
}


template MyRecordType t2 := {
field1 := "foo",
field2 := 1,
field3 := true
}
```

(continues on next page)

(continued from previous page)

```
template MyRecordType t3 := {
field1 := "bar",
field2 := 1,
field3 := true
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- An approach to quality engineering of TTCN-3 test specifications
- Pattern-based Smell Detection in TTCN-3 Test Suites
- Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites

1.1.1.14 Duplicate Test Code

Definition:

- Another test method tests the same thing.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- What We Know About Smells in Software Test Code

1.1.1.15 Duplicate Test Runs

Definition:

- When the inherited test methods are executed twice: once in the parent test case (i.e., a non-abstract class) and once in the child test case.

Code Example:

- No code examples yet…

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Revisiting Test Impact Analysis in Continuous Testing From the Perspective of Code Dependencies

1.1.1.16 Duplicated Actions

Definition:

- Some duplicated actions appear in multiple places.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Bad smells and refactoring methods for GUI test script

1.1.1.17 Duplicated Code In Conditional

Definition:

- The duplicated code can appear in a series of conditionals (with different conditions and the same action in each check) or in all legs of a conditional

Code Example:

```
function checkSomething(in float p1, in float p2) return boolean {
    if (p1 < 0.0) {
        return false;
    }
    if (p2 >= 7.0) {
        return false;
    }
    if (p2 < p1) {
        return false;
    }
    return true;
}
```

(continues on next page)

(continued from previous page)

```
function checkSomethingElse(in float p1) runs on ExampleComponent {
    var charstring result;
    if (p1 > 0) {
        result := "foo";
        pt.send(result);
    } else {
        result := "bar";
        pt.send(result);
    }
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- An approach to quality engineering of TTCN-3 test specifications
- Pattern-based Smell Detection in TTCN-3 Test Suites
- Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites

1.1.1.18 Duplicated Code

Definition:

- If you have duplicated code in tests, it makes it harder to refactor the implementation code because you have a disproportionate number of tests to update. Tests should help you refactor with confidence, rather than be a large burden that impedes your work on the code being tested.

Code Example:

```
assertEqual('Joe', person.getFirstName())
assertEqual('Bloggs', person.getLastName())
assertEqual(23, person.getAge())
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- A Refactoring Tool for TTCN-3

- An Empirical Study into the Relationship Between Class Features and Test Smells
- Is duplicated code more tolerable in unit tests?
- Test Smell Detection Tools: A Systematic Mapping Study
- TestQ: Exploring Structural and Maintenance Characteristics of Unit Test Suites
- What We Know About Smells in Software Test Code
- Writing good gherkin

1.1.1.19 Half A Helper Method

Definition:

- Where there's a utility method to help a test do its job, yet all calls to it are immediately followed by the exact same code. This is because the method is only doing half the job it should, so your test has more boilerplate in it.

Code Example:

```
function readFile(filename) {
// open file and return its string
}

function substituteId(file, id) {
const placeholder = 'REPLACE-ME';

// do global replace of the placeholder with id
// return that
}

// call site
const file = readFile('foo.txt');
const substituted = substituteId(file, currentId);
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.1.1.20 Missing Parameterised Test

Definition:

- When you did it the long way round because you didn't bring in parameterisation

Code Example:

```
@ParameterizedTest
@CsvSource({
"a,A",
"b,B",
"bbb,BBB",
"bBbB,BBBB"
})
void capitalizerTurnsInputToCapitals(String input, String expected) {
    assertThat(Capitalizer.toCapitals(input))
        .isEqualTo(expected);
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.1.1.21 Missing Test Data Factory

Definition:

- Where every test has its own way of making the same test example data

Code Example:

```
import { createMockSession, createMockUser } from
'../../components/auth/__tests__/User.test.ts';
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.1.1.22 Multiple Tests Testing The Same Or Similar Things

Definition:

- Multiple tests are created that have the same test code, but change the values passed

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-Patterns In Unit Testing

1.1.1.23 Reinventing The Wheel

Definition:

- While Cut and Paste Code Reuse deliberately makes copies of existing code to reduce the effort of writing tests, it is also possible to accidently write the same sequence of statements in different tests.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- xUnit test patterns: Refactoring test code

1.1.1.24 Second Class Citizens

Definition:

- Test code isn't as well refactored/structured as production code, containing a lot of duplicated code, making it hard to maintain tests.

Also Known As:

- Test Code Duplication

References:

Quality attributes

- - Code Example
- - Cause and Effect

- - Frequency
- - Refactoring

- Anti-Patterns - Digital Tapestry
- Smells in software test code: A survey of knowledge in industry and academia
- Unit testing Anti-patterns catalogue
- Unveiling 6 Anti-Patterns in React Test Code: Pitfalls to Avoid

1.1.1.25 Test Clones

Definition:

- Tests contain similar parts (e. g., introduced by copy&paste)

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Generating refactoring proposals to remove clones from automated system tests
- Hunting for smells in natural language tests

1.1.1.26 Test Code Duplication

Definition:

- This test smell normally identifies classes that contain test methods with repeated test code steps.

Also Known As:

- Second Class Citizen

Code Example:

```
class TestFlight(unittest.TestCase):
    def test_mileage_init(self):
        airLine = '2569'
        mileage = 1000
        flight = Flight(airLine,mileage)
        self.assertEqual(flight.mileage,1000)

    def test_fuel_is_full(self):
        airLine = '2569'
        mileage = 1000
        flight = Flight(airLine,mileage)
        self.assertTrue(flight.fullFuel)
```

(continues on next page)

(continued from previous page)

```
    def test_is_valid_air_line_code(self):
        airLine = '2569'
        mileage = 1000
        flight = Flight(airLine,mileage)
        self.assertTrue(flight.isValidAirLineCode())

if __name__ == '__main__':
    unittest.main(argv=['first-arg-is-ignored'], exit=False)
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- A survey on test practitioners' awareness of test smells
- An empirical analysis of the distribution of unit test smells and their impact on software maintenance
- An exploratory study of the relationship between software test smells and fault-proneness
- Are test smells really harmful? An empirical study
- Automatic generation of smell-free unit tests
- Categorising Test Smells
- Did You Remember To Test Your Tokens?
- Enhancing developers' awareness on test suites' quality with test smell summaries
- How are test smells treated in the wild? A tale of two empirical studies
- Inspecting Automated Test Code: A Preliminary Study
- LCCSS: A Similarity Metric for Identifying Similar Test Code
- On the diffusion of test smells in automatically generated test code: an empirical study
- On the interplay between software testing and evolution and its effect on program comprehension
- Refactoring Test Code
- Refactoring Test Smells With JUnit 5: Why Should Developers Keep Up-to-Date?
- Refactoring Test Smells: A Perspective from Open-Source Developers
- Test Artifacts — The Practical Testing Book
- Test Smell Detection Tools: A Systematic Mapping Study
- Test code quality and its relation to issue handling performance
- TestAXE: Automatically Refactoring Test Smells Using JUnit 5 Features
- xUnit test patterns: Refactoring test code

1.1.1.27 Test Redundancy

Definition:

- Occurs when the removal of a test does not impact the effectiveness of the test suite

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Automatic generation of smell-free unit tests
- Improving Student Testing Practices through a Lightweight Checklist Intervention.
- Machine Learning-Based Test Smell Detection
- Test Smell Detection Tools: A Systematic Mapping Study

1.1.1.28 The First And Last Rites

Definition:

- Where there's some ritual/boilerplate at the start and end of most test bodies, suggesting a lack of common setup/teardown code

Also Known As:

- Oops I Forgot The Setup

Code Example:

```
@Test
public void connectionWorks() {
    database = openDatabase();

    database.healthCheck();

    database.close();
}

@Test
public void countRows() {
    database = openDatabase();

    assertThat(database.countAll())
    .isEqualTo(0);

    database.close();
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.1.1.29 Two For The Price Of One

Definition:

- Sometimes a sign of missing parameterised tests – where a single test case is testing two use cases with the same set up

Code Example:

```
private LocalDateTime now = LocalDateTime.now();
private LocalDateTime tomorrow = LocalDateTime.plus(1, DAYS);
private LocalDateTime yesterday = LocalDateTime.minus(1, DAYS);

@Test
public void dateComparisonWorksForPastAndFuture() {
    assertDateComparison(now, tomorrow, "is in the future");
    assertDateComparison(now, yesterday, "is in the past");
}

private void assertDateComparison(LocalDateTime baseline,
                                  LocalDateTime compare,
                                  String expected) {
    assertEquals(expected, MyDates.describe(baseline, compare);
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.1.1.30 Very Similar Test Cases

Definition:

- Copy-pasting many similar chunks of test code to add new test cases;

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-patterns of automated testing

1.1.2 Complex - Hard to understand

1.1.2.1 Absence Of Why

Definition:

- where the code of the test just IS and does nothing to explain the use case

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.1.2.2 Bad Comment Rate

Definition:

- The comment rate (number of comments per line) is too high or too low.

Code Example:

```
function ptc_CC_PR_MP_RQ_V_030(CSeq lo cCSeq s) runs on SipComponent {
  var Request vINVITERequest;
  var Request vBYERequest;
  var Request vACKRequest;
  charstring vbranch := "";
  initPTC(locCSeq s);
  vDefault := activate(defaultCCPRPTC());
```

(continues on next page)

(continued from previous page)

```
  alt {
    [] SIPP.receive(INVITERequest r1)->
    value vINVITERequest sendt_label {
    TGuard.stop;
    setHeadersOnReceiptOfInvite(vINVITERequest);
    sendPTC200OKInvite();
    setverdict(pass);
    repeat;
    }
    [] SIPP.receive(ACKRequest r1(vCallId))-> value vACKRequest sendt_label {
      vVia := vACKRequest.msgHeader.via;
      if (ispresent(vVia.viaBody[0].viaParams)) {
        var SemicolonParamList tmp_params := vVia.viaBody[0].viaParams;
        if (checkBranchPresent(tmp_params, vbranch)) {
          if (match(v_branch, ValidBranch)) {
            setverdict(pass);
          } else {
            setverdict(fail);
          };
        } else {
          setverdict(fail)
        };
      } else {
        setverdict(fail)
      };
      cpA.send(CMCheckDone);
      repeat;
    }
    [] SIPP.receive(BYERequest r1(vCallId))-> value vBYERequest sendt_label {
      setHeadersOnReceiptO fBye(vBYERequest);
      send200OK();
    }
    [] cpA.receive(CMStop) {
      all timer.stop;
      stop;
    }
    [] SIPP.receive {
      repeat;
    }
    [] TGuard.timeout {
      setverdict(fail);
      stop;
    }
  }
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect

- - Frequency
- - Refactoring

- An approach to quality engineering of TTCN-3 test specifications
- Pattern-based Smell Detection in TTCN-3 Test Suites
- Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites

1.1.2.3 Boilerplate Hell

Definition:

- where you can't read the test because there's so much code, perhaps a case of missing test data factory

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.1.2.4 Comments

Definition:

- When something needs to be explained, it is probably better to move into a method with a descriptive name

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- A Refactoring Tool for TTCN-3

1.1.2.5 Complicated Logic In Tests

Definition:

- Complicated logic within the tests themselves

Code Example:

- No code examples yet…

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Unveiling 6 Anti-Patterns in React Test Code: Pitfalls to Avoid

1.1.2.6 Hard-Coded Test Data

Definition:

- Data values in the fixture, assertions or arguments of the SUT are hard-coded in the Test Method obscuring cause-effect relationships between inputs and expected outputs.

Code Example:

```java
public void testAddItemQuantity_severalQuantity_v12(){
    //  Setup Fixture
    Customer cust = createACustomer(new BigDecimal("30"));
    Product prod = createAProduct(new BigDecimal("19.99"));
    Invoice invoice = createInvoice(cust);
    // Exercise SUT
    invoice.addItemQuantity(prod, 5);
    // Verify Outcome
    LineItem expected = new LineItem(invoice, prod, 5,
            new BigDecimal("30"), new BigDecimal("69.96"));
    assertContainsExactlyOneLineItem(invoice, expected);
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Detecting redundant unit tests for AspectJ programs
- Obscure Test

1.1.2.7 Hard-Coded Values

Definition:

- Scalar values or value objects that are used directly in fixture setup, as parameters in the test exercise or as expected values in the verification. That is, they are not assigned to a named constant or variable.

Code Example:

```
public function testGetTranslationFileTimestamp()
{
    $this->fileManagerMock->expects($this->once())
        ->method('getTranslationFileTimestamp')
        ->willReturn(1445736974);
    $this->assertEquals(1445736974, $this->model->getTranslationFileTimestamp());
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Hunting for smells in natural language tests
- Test Smell: Hard Coded Values

1.1.2.8 Hard-To-Test Code

Definition:

- Legacy software (any software that doesn't have a complete suite of automated tests) can be hard to test since we are typically writing the tests "last" (after the software already exists.) If the design of the software is not conducive to automated testing, we may be forced to use Indirect Testing (see Obscure Test) via awkward interfaces that involve a lot of accidental complexity; that may result in Fragile Tests or a Fragile Fixture.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Automatic generation of smell-free unit tests
- Enhancing developers' awareness on test suites' quality with test smell summaries
- xUnit test patterns: Refactoring test code
- xUnit test patterns: Refactoring test code
- xUnit test patterns: Refactoring test code
- xUnit test patterns: Refactoring test code

1.1.2.9 Hard-To-Write Test

Definition:

- Making hacks that need an explanation or having complicated setups, so that a scenario can be tested

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-patterns of automated testing

1.1.2.10 Hardcoded Environment Configuration

Definition:

- Imagine same checks must be run against both Firefox and Chrome; or against local and then pre-production test environments. No way to do it if you hardcoded references to browser type, server host or databases.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-patterns in test automation

1.1.2.11 Hardcoded Environment

Definition:

- The test contains hardcoded references to the environment when the same requirement must be run against different test environments instead of having an environment-agnostic test.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-patterns in test automation
- On the Maintenance of System User Interactive Tests
- Smells in System User Interactive Tests

1.1.2.12 Hardcoded Literals

Definition:

- Hardcoding number of sync tasks

Code Example:

- No code examples yet…

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Development Patterns and Anti-Patterns - Medium

1.1.2.13 Hardcoded Test Data

Definition:

- maintaning hardcoded test data becomes a nightmare

Also Known As:

- Magic Strings

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-patterns in test automation

1.1.2.14 Herp Derp

Definition:

- Words and comments in test code or names that add nothing, like simple or test or //given

Code Example:

```
1. Calling the test method test in any way - we know it’s a test, get on with it
2. Weak adjective expressions like simple - what makes it simple?
3. Commenting a line of code with a description of what that line’s implementation is␣
↪doing - e.g. // assert that it's true - we can see what it’s doing... WHY is it doing␣
↪it?
4. Pasting in the same comments from other tests whether or not they’re relevant - this␣
↪is probably a case for reducing boilerplate so you don’t need as much paste, or as␣
↪much comment
5. Naming test inputs and outputs after their type, rather than their purpose in the␣
↪test - e.g. String string1 = code.getUserName()
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.1.2.15 Hidden Complexity

Definition:

- A piece of code that uses innocent looking API that in fact performs a lot of computation.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Parameterized Test Patterns For Effective Testing with Pex

1.1.2.16 It Looks Right To Me

Definition:

- Where the test data for negative cases makes the test hard to understand

Code Example:

```
expect(getFieldCaseInsensitive(obj, 'username'))
.toEqual('Mr User');

expect(getFieldCaseInsensitive(obj, 'Username'))
.toEqual('Mr User');

expect(getFieldCaseInsensitive(obj, 'UserNaME'))
.toEqual('Mr User');
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.1.2.17 Keyword Driven Testing

Definition:

- Keyword Driven Testing (KDT) is a remnant from the early to mid-2000s when QTP aka UFT was popular. At one point, we believed that somehow we can write test automation code in such a way that manual testers will be able to string together a bunch of keywords aka functions to make tests. We used Excel sheets to design test cases by stringing together a bunch of functions

Code Example:

- No code examples yet…

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Top 17 Automated Testing Best Practices (Supported By Data)

1.1.2.18 Lack Comments

Definition:

- This smell comes from the lack of comments in the test code, which makes a test case hard to understand

Code Example:

- No code examples yet…

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Improving Student Testing Practices through a Lightweight Checklist Intervention.

1.1.2.19 Large Macro Component

Definition:

- A single macro component contains too many primitive components, macro events, and macro components

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Bad smells and refactoring methods for GUI test script

1.1.2.20 Large Module

Definition:

- indicates that the module may be trying to do too much. It becomes hard to read and it likely combines too many distinct code parts which are better off in multiple modules

Also Known As:

- Large Class

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- A Refactoring Tool for TTCN-3

1.1.2.21 Large Test File

Definition:

- Things can be hard to find in a large test file and its documentation value is lost.

References:

Quality attributes

- - Code Example
- - Cause and Effect

- - Frequency
- - Refactoring

- Smells of Testing (signs your tests are bad)

1.1.2.22 Long Class

Definition:

- To properly execute your automated test, your code must open your application in a browser, perform the necessary actions on the application, and then verify the resulting state of the application. While these are all needed for your test, these are different responsibilities, and having all of these responsibilities together within a single class is a code smell. This smell makes it difficult to grasp the entirety of what's contained within the class, which can lead to redundancy and maintenance challenges.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Writing good gherkin

1.1.2.23 Long Function

Definition:

- Long functions are hard to understand and indirection can be better supported by small functions

Also Known As:

- Long Method

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- A Refactoring Tool for TTCN-3

1.1.2.24 Long Macro Event

Definition:

- A macro event (or keyword) contains too many actions.

Also Known As:

- Long Keyword

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Bad smells and refactoring methods for GUI test script

1.1.2.25 Long Method

Definition:

- It's where a method or function has too many responsibilities. Many times a method does not start off being too long, but over time it grows and grows, taking on additional responsibilities. This poses an issue when tests want to call into a method to do one thing, but will ultimately have multiple other things executed as well.

Also Known As:

- Long Function

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Writing good gherkin

1.1.2.26 Long Statement Block

Definition:

- Long statement block in function, test case or altstep. A long function is more difficult to understand than a short one. Although the use of short functions (i.e. methods) is especially important for modern objectoriented languages, short functions have a certain importance for TTCN-3 as well. Long statement blocks in functions, test cases and altsteps should be decomposed into short functions with meaningful names.

Code Example:

```
function ptc_CC_PR_MP_RQ_V_030(CSeq lo cCSeq s) runs on SipComponent {
  var Request vINVITERequest;
  var Request vBYERequest;
  var Request vACKRequest;
  charstring vbranch := "";
  initPTC(locCSeq s);
  vDefault := activate(defaultCCPRPTC());
  alt {
    [] SIPP.receive(INVITERequest r1)->
    value vINVITERequest sendt_label {
    TGuard.stop;
    setHeadersOnReceiptOfInvite(vINVITERequest);
    sendPTC200OKInvite();
    setverdict(pass);
    repeat;
    }
    [] SIPP.receive(ACKRequest r1(vCallId))-> value vACKRequest sendt_label {
      vVia := vACKRequest.msgHeader.via;
      if (ispresent(vVia.viaBody[0].viaParams)) {
        var SemicolonParamList tmp_params := vVia.viaBody[0].viaParams;
        if (checkBranchPresent(tmp_params, vbranch)) {
          if (match(v_branch, ValidBranch)) {
            setverdict(pass);
          } else {
            setverdict(fail);
          };
        } else {
          setverdict(fail)
        };
      } else {
        setverdict(fail)
      };
      cpA.send(CMCheckDone);
      repeat;
    }
    [] SIPP.receive(BYERequest r1(vCallId))-> value vBYERequest sendt_label {
      setHeadersOnReceiptO fBye(vBYERequest);
      send200OK();
    }
    [] cpA.receive(CMStop) {
      all timer.stop;
      stop;
    }
```

(continues on next page)

(continued from previous page)

```
    [] SIPP.receive {
      repeat;
    }
    [] TGuard.timeout {
      setverdict(fail);
      stop;
    }
  }
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- An approach to quality engineering of TTCN-3 test specifications
- Pattern-based Smell Detection in TTCN-3 Test Suites
- Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites

1.1.2.27 Long Test Steps

Definition:

- One or many test steps are very long, performing a lot of actions on the system under test.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Hunting for smells in natural language tests
- On the Maintenance of System User Interactive Tests
- Smells in System User Interactive Tests

1.1.2.28 Long Test

Definition:

- A Long Test is a test that consists of lot of code and statements. Such tests are mostly (but not necessarily) complex and badly document the purpose of the test and the application code. Furthermore they tend to test too much functionality, maybe even getting eager.

Also Known As:

- Obscure Test

Code Example:

```
# Subject under test
class Stack
  def initialize
    @items = []
  end

  def push(item)
    @items << item
  end

  def pop
    @items.pop
  end

  def peek
    @items.last
  end

  def depth
    @items.size
  end
end

# Test
class Long < SmellTest
  def test_make_sure_everything_works
    subject = Stack.new

    # Test Push
    subject.push("A")
    subject.push("B")
    subject.push("C")

    assert_equal "C", subject.pop
    assert_equal "B", subject.pop
    assert_equal "A", subject.pop

    # Test Peek
    subject.push("D")
    subject.push("E")
```

(continues on next page)

(continued from previous page)

```
    assert_equal "E", subject.peek

    subject.pop

    assert_equal "D", subject.peek

    subject.pop

    # Test Depth
    subject.push("F")
    subject.push("G")

    assert_equal 2, subject.depth

    # Test Pop
    subject.pop
    subject.pop

    assert_equal 0, subject.depth
    assert_equal nil, subject.pop
  end
end
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- A workbook repository of example test smells and what to do about them
- Assessing test quality - TestLint
- Categorising Test Smells
- Detection of test smells with basic language analysis methods and its evaluation
- Rule-based Assessment of Test Quality
- Test Smell Detection Tools: A Systematic Mapping Study

1.1.2.29 Long/Complex/Verbose/Obscure Test

Definition:

- It is difficult to understand the test at a glance. The test usually has excessive lines of code.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- What We Know About Smells in Software Test Code

1.1.2.30 Magic Number Test

Definition:

- This smell occurs when a test method contains unexplained and undocumented numeric literals as parameters or as values to identifiers. These magic values do not sufficiently indicate the meaning/purpose of the number. Hence, they hinder code understandability. Consequently, they should be replaced with constants or variables, thereby providing a descriptive name for the value.

Code Example:

```
@Test
public void testGetLocalTimeAsCalendar() {
    Calendar localTime = calc.getLocalTimeAsCalendar(BigDecimal.valueOf(15.5D), Calendar.
→getInstance());
    assertEquals(15, localTime.get(Calendar.HOUR_OF_DAY));
    assertEquals(30, localTime.get(Calendar.MINUTE));
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- A survey on test practitioners' awareness of test smells
- An Exploratory Study on the Refactoring of Unit Test Files in Android Applications
- Analyzing Test Smells Refactoring from a Developers Perspective
- Automatic Identification of High-Impact Bug Report by Product and Test Code Quality

- Automatic generation of smell-free unit tests
- Characterizing High-Quality Test Methods: A First Empirical Study
- Handling Test Smells in Python: Results from a Mixed-Method Study
- How are test smells treated in the wild? A tale of two empirical studies
- Investigating Severity Thresholds for Test Smells
- Investigating Test Smells in JavaScript Test Code
- On the Distribution of "Simple Stupid Bugs" in Unit Test Files: An Exploratory Study
- On the diffusion of test smells and their relationship with test code quality of Java projects
- On the distribution of test smells in open source Android applications: an exploratory study
- On the influence of Test Smells on Test Coverage
- On the test smells detection: an empirical study on the jnose test accuracy
- On the use of test smells for prediction of flaky tests
- PyNose: A Test Smell Detector For Python
- Pytest-Smell: a smell detection tool for Python unit tests
- Refactoring Test Smells: A Perspective from Open-Source Developers
- Software Unit Test Smells
- Test Artifacts — The Practical Testing Book
- Test Smell Detection Tools: A Systematic Mapping Study
- The secret life of test smells-an empirical study on test smell evolution and maintenance
- What the Smell? An Empirical Investigation on the Distribution and Severity of Test Smells in Open Source Android Applications
- tsDetect: an open source test smells detection tool

1.1.2.31 Magic Values

Definition:

- Magic Values are literals not defined as constant. Numeric literals are called Magic Numbers, string literals are called Magic Strings.

Code Example:

```
testcase SIP_CC_PR_TR_SE_TI_004 (inout CSeq loc_CSeq_s, CSeq loc_CSeq_ptcs)
  runs on SipComponent system SipInterfaces
{
    var SipComponent vptc;
    var Response vResponse;
    var float vdelay;
    vDefault := activate(defaultCCPR());
    vptc := SipComponent.create;
    initConfig1(mtc, vptc, system);
    initMTCphase1(loc_CSeq_s);
    setHeadersPtcInvite(loc_CSeq_s);
    vptc.start(ptcWaitCheckInviteCompletedState(loc_CSeq_ptcs));
```

(continues on next page)

(continued from previous page)

```
    initMTCphase2();
    SIPP.send(INVITE Request s2(vRequestUri, vCallId, loc_CSeq_s, vFrom, vTo, vVia)) to␣
↪sentlabel;
    vCSeq := loc_CSeq_s;
    awaitingFirstAnyFinalResp(vResponse, loc_CSeq_s);
    setHeadersOnReceiptOfResponse(loc_CSeq_s, vResponse);
    // First Repetition
    repeatRespInTime(vResponse, loc_CSeq_s, PXT1 * 1.5);
    // Second Repetition
    vdelay := minValue(2.0 * PXT1, PXT2) * 1.5;
    repeatRespInTime(vResponse, loc_CSeq_s, vdelay);
    // Third repetition
    vdelay := minValue(4.0 * PXT1, PXT2) * 1.1;
    repeatRespInTime(vResponse, loc_CSeq_s, vdelay);
    sendACK(loc_CSeq_s);
    synchroniseCheckDone();
    waitendptc(vptc);
} // end testcase SIP_CC_PR_TR_SE_TI_004
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- An approach to quality engineering of TTCN-3 test specifications
- Pattern-based Smell Detection in TTCN-3 Test Suites
- Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites

1.1.2.32 Obscure Test

Definition:

- It is difficult to understand the test at a glance. Automated tests should serve at least two purposes. First, they should act as documentation of how the system under test (SUT) should behave; we call this Tests as Documentation (see Goals of Test Automation on page X). Second, they should be a self-verifying executable specification. These two goals are often contradictory because the level of detailed needed for tests to be executable may make the test so verbose as to be difficult to understand.

Also Known As:

- Long Test, Complex Test, Verbose Test

Code Example:

```
context 'the element does not exist' do
before do
```

(continues on next page)

(continued from previous page)

```
    contents = %(
    <?xml version="1.0" encoding="UTF-8"?>
    <rss version="2.0" xmlns:atom="http://www.w3.org/2005/Atom" xmlns:content="http://
↪purl.org/rss/1.0/modules/content/" xmlns:itunes="http://www.itunes.com/dtds/podcast-1.
↪0.dtd">
        <channel>
        <item></item>
        </channel>
    </rss>
    )

    xml_doc = Nokogiri::XML(contents)
    episode_element = xml_doc.xpath('//item').first
    @rss_feed_episode = RSSFeedEpisode.new(episode_element)
end

it 'returns an empty string' do
    expect(@rss_feed_episode.content('title')).to eq('')
end
end
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- An analysis of information needs to detect test smells
- Detecting redundant unit tests for AspectJ programs
- Did You Remember To Test Your Tokens?
- Enhancing developers’ awareness on test suites’ quality with test smell summaries
- Inspecting Automated Test Code: A Preliminary Study
- Obscure Test
- On the Maintenance of System User Interactive Tests
- Rails Testing Antipatterns
- Smells in System User Interactive Tests
- Test code quality and its relation to issue handling performance
- Test smell: Obscure Test
- xUnit test patterns: Refactoring test code
- xUnit test patterns: Refactoring test code

1.1.2.33 Optimizing Dry

Definition:

- Pursuing DRYness by creating variables and functions in tests;

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-patterns of automated testing

1.1.2.34 Over Refactoring Of Tests

Definition:

- where you can't read them because they've been DRYed out to death

Code Example:

```
// Over Refactoring Of Test
assertThat(calculateAnswer(INPUT))
 .isEqualTo(EXPECTED);
```

```
//before
assertThat(calculateAnswercountTheWordsIn("This is a string"))
 .isEqualTo(4);
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.1.2.35 Overcommented Test

Definition:

- Overcommented Tests dene too many comments, obfuscating the code and distracting from the purpose of the test.

Code Example:

```
DebuggerTest >> #testUnwindDebuggerWithStep
  "test if unwind blocks work properly when a debugger is closed"
  | sema process debugger top |
  sema := Semaphore forMutualExclusion.
  self assert: sema isSignaled.
  process := [sema critical:[sema wait]]
  forkAt: Processor userInterruptPriority.
  self deny: sema isSignaled.
  "everything set up here - open a debug notifier"
  debugger := Debugger openInterrupt: 'test' onProcess: process.
  "get into the debugger"
  debugger debug.
  top := debugger topView.
  "set top context"
  debugger toggleContextStackIndex: 1.
  "do single step"
  debugger doStep.
  "close debugger"
  top delete.
  "and see if unwind protection worked"
  self assert: sema isSignaled
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Assessing test quality - TestLint
- Automatic generation of smell-free unit tests
- Categorising Test Smells
- Rule-based Assessment of Test Quality
- Test Smell Detection Tools: A Systematic Mapping Study

1.1.2.36 Overly Complex Tests

Definition:

- Unit tests, like production code, should be easily understandable. In general, a programmer must understand the intent of an individual test as fast as possible. If a test is so complicated that you can't immediately tell if it is correct or not, it makes it very difficult to determine if the cause of a test failure is bad production code or bad test code. Even worse, this allows the possibly of code that incorrectly passes a test.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- JUnit Anti-patterns

1.1.2.37 Overly Dry Tests

Definition:

- DRY (Don't Repeat Yourself) is a good idea, but pulling out all repetition can lead to some very hard to understand tests.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Categorising Test Smells
- Smells of Testing (signs your tests are bad)

1.1.2.38 Overuse Of Abstractions

Definition:

- Because test code is documentation it needs to be descriptive and easy to follow. Instead of DRY test code should be DAMP (Descriptive And Meaningful Phrases). Since the goal is to understand the test and the code you are testing some repetition may be necessary.

Also Known As:

- it's too DRY

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-Patterns In Unit Testing

1.1.2.39 Self Important Test Data

Definition:

- Test data is more complex than is needed to exercise some behavior in a test.

Code Example:

```ruby
# Subject under test
def gps_within_location?(gps, location)
  boundaries = boundaries_for_zip(location.zip)

  boundaries.northWest.lat >= gps.lat &&
    boundaries.southEast.lat <= gps.lat &&
    boundaries.northEast.lng >= gps.lng &&
    boundaries.southWest.lng <= gps.lng
end

# Test
class SelfImportantTestData < SmellTest
  def test_gps_inside_location
    gps = OpenStruct.new(
      altitude: 3000,
      course: 3.62,
      horizontalAccuracy: 10,
      lat: 43,
      lng: -77,
      secondsSinceLastUpdate: 4,
      speed: 3,
      utcOfLastFix: 180145,
      verticalAccuracy: 15
    )
    location = OpenStruct.new(
      name: 'Cup O Joe',
      streetLine1: '8312 Mulberry St',
      streetLine2: 'Lot #326 c/o very detailed test data',
      city: 'Grandview Heights',
      state: 'OH',
      stateFullName: 'Ohio',
      zip: 43221,
      zipPlus4: 8312
```

(continues on next page)

(continued from previous page)

```
    )

    result = gps_within_location?(gps, location)

    assert_equal true, result
  end

  def test_gps_not_inside_location
    gps = OpenStruct.new(
      altitude: 4000,
      course: 14.18,
      horizontalAccuracy: 5,
      lat: 48,
      lng: -77,
      secondsSinceLastUpdate: 2,
      speed: 1,
      utcOfLastFix: 141445,
      verticalAccuracy: 25
    )
    location = OpenStruct.new(
      name: 'J.F.K Elementary School',
      streetLine1: '1438 Soledad St',
      streetLine2: nil,
      city: 'Columbus',
      state: 'OH',
      stateFullName: 'Ohio',
      zip: 43221,
      zipPlus4: 4294
    )

    result = gps_within_location?(gps, location)

    assert_equal false, result
  end
end

# Fake production implementations to simplify example test of subject
def boundaries_for_zip(zip)
  OpenStruct.new(
    northWest: OpenStruct.new(lat: 45, lng: -80),
    southWest: OpenStruct.new(lat: 40, lng: -80),
    southEast: OpenStruct.new(lat: 40, lng: -75),
    northEast: OpenStruct.new(lat: 45, lng: -75)
  )
end
```

References:

Quality attributes

- - Code Example
- - Cause and Effect

- - Frequency
- - Refactoring

- A workbook repository of example test smells and what to do about them

1.1.2.40 Tests Are Difficult To Write

Definition:

- If you have an exorbitantly difficult time writing tests, even as a newbie, you should take a critical look at the code you're trying to test. It usually indicates excessive coupling.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Unit Testing Smells: What Are Your Tests Telling You?

1.1.2.41 Using Complicated Data Store

Definition:

- The logic to read and manage Excel adds extra overhead to your test automation that isn't necessary. You will need to write 100s of lines of code just to manage an Excel object and read data based on column headers and row locations. It's not easy and is prone to error.

Code Example:

- No code examples yet…

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Top 17 Automated Testing Best Practices (Supported By Data)

1.1.2.42 Using Complicated X-Path Or Css Selectors

Definition:

- Using element identification selectors that have long chains from the DOM in them leads to fragile tests, as any change to that chain in the DOM will break your tests.

Code Example:

```
private static readonly By TeaTypeSelector =
          By.CssSelector(
              '#input-tea-type < div < div.TeaSearchRow < div.TeaSearchCell.no < div:nth-
↪child(2) < label');
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Five automated acceptance test anti-patterns

1.1.2.43 Using Test Case Inheritance To Test Source Code Polymorphism

Definition:

- Developers may implement inheritance in test cases to test source code polymorphism so that code duplication of common test methods can be reduced (i.e., following the DRY principle – "Don't Repeat Yourself"). However, sometimes the unnecessary inheritance relationship in test cases may increase the difficulty of test maintenance and make test evolution more error-prone.

Code Example:

- No code examples yet…

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Revisiting Test Impact Analysis in Continuous Testing From the Perspective of Code Dependencies

1.1.2.44 Verbose Test

Definition:

- A test method with more than 30 lines. It may indicate that the method has several responsibilities, impacting the test method maintenance.

Also Known As:

- Obscure Test

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Automatic Identification of High-Impact Bug Report by Product and Test Code Quality
- Automatic generation of smell-free unit tests
- Handling Test Smells in Python: Results from a Mixed-Method Study
- Investigating Severity Thresholds for Test Smells
- On the diffusion of test smells and their relationship with test code quality of Java projects
- On the influence of Test Smells on Test Coverage
- On the test smells detection: an empirical study on the jnose test accuracy
- On the use of test smells for prediction of flaky tests
- Smart prediction for refactorings in the software test code
- TEMPY: Test Smell Detector for Python
- Test Smell Detection Tools: A Systematic Mapping Study
- TestQ: Exploring Structural and Maintenance Characteristics of Unit Test Suites

1.1.2.45 What Are We Testing?

Definition:

- where the test data, or the way we produce it, is not self-explanatory for the use case. This is the general case of many of the below smells, but also includes how test data is represented in the code.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency

- - Refactoring

- Test Smells - The Coding Craftsman

1.1.3 In association with production code

1.1.3.1 Behavior Sensitivity

Definition:

- Behavior Sensitivity is when changes to the SUT cause other tests to fail.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Categorising Test Smells
- xUnit test patterns: Refactoring test code

1.1.3.2 Changing Implementation To Make Tests Possible

Definition:

- Changing the implementation solely for the sake of making testing possible

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-patterns of automated testing

1.1.3.3 Code Pollution

Definition:

- It takes place when you introduce additional code to your main code base in order to enable unit testing

Code Example:

```
public class OrderRepository
{
    public void Save(Order order)
    {
        /* ... */
    }

    public Order GetById(long id)
    {
        /* ... */
    }
}
```

```
[Fact]
public void Some_integration_test()
{
    // Arrange
    var repository = new OrderRepository();
    var service = new OrderService(repository);
    long customerId = 42;

    // Act
    service.DoSomething(customerId);

    // Assert
    IReadOnlyList<Order> orders = repository.GetByCustomerId(customerId); // Code added␣
↪for the use in this unit test
    /* validate orders */
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Code pollution

1.1.3.4 Code Run Only By Tests

Definition:

- Dead code (never touched in production) is very confusing to the developer trying to understand the system. Tested dead code doubly so.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Smells of Testing (signs your tests are bad)

1.1.3.5 Context Logic In Production Code

Definition:

- When the production code becomes aware of the context in which it is used.

Code Example:

```
public static void SaveToDatabase(Customer customerToWrite)
{
  if (AreWeTesting)
    WriteWithMockDatabase(customerToWrite);
  else
    Write(customerToWrite);
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Unit Testing Smells: What Are Your Tests Telling You?

1.1.3.6 Equality Pollution

Definition:

- The code that is put into production contains logic that should be exercised only during tests… A system that behaves one way in the test lab and an entirely different way in production is a recipe for disaster!

Code Example:

```
/// <summary>
/// Verifies that two objects are equal, using a default comparer.
/// </summary>
/// <typeparam name="T">The type of the objects to be compared</typeparam>
/// <param name="expected">The expected value</param>
/// <param name="actual">The value to be compared against</param>
/// <exception cref="EqualException">Thrown when the objects are not equal</exception>
public static void Equal<T>(T expected, T actual)
{
    Equal(expected, actual, GetEqualityComparer<T>());
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- How to Compare Object Instances in your Unit Tests Quickly and Easily
- xUnit test patterns: Refactoring test code

1.1.3.7 Fire And Forget

Definition:

- A test that's at risk of exiting prematurely, typically before its assertions can run and fail the test run if necessary

Also Known As:

- Plate-Spinning

Code Example:

```
# Subject under test
def load_user(id)
  path = "/users/#{id}"

  get(path) { |er, user|
    user.resolved_via = path
    yield er, user if block_given?
  }
```

(continues on next page)

(continued from previous page)

```
end

# Test
class FireAndForget < SmellTest
  include UnreliableMinitestPlugin

  def test_gets_user_and_decorates_path
    load_user(42) { |er, user|
      assert_equal "/users/42", user.resolved_via
      assert_equal "Jo", user.name
    }
  end
end

# Fake production implementations to simplify example test of subject
def get(path)
  Thread.new do
    sleep 0.01 if rand < 0.5 # sometimes it takes time
    yield nil, OpenStruct.new(name: "Jo") if block_given?
  end
end
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- A workbook repository of example test smells and what to do about them
- Smells in software test code: A survey of knowledge in industry and academia
- Toward static test flakiness prediction: a feasibility study

1.1.3.8 For Testers Only

Definition:

- This smell arises when a production class contains methods only used by test methods. This kind of production classes should be removed, since it does not provide functionalities used by other classes in the system. From the testing side, this smell involves an extra effort needed in order to comprehend and modify as assertions.

References:

Quality attributes

- - Code Example
- - Cause and Effect

- - Frequency
- - Refactoring

- A survey on test practitioners' awareness of test smells
- An empirical analysis of the distribution of unit test smells and their impact on software maintenance
- An exploratory study of the relationship between software test smells and fault-proneness
- Are test smells really harmful? An empirical study
- Automatic generation of smell-free unit tests
- Categorising Test Smells
- Enhancing developers' awareness on test suites' quality with test smell summaries
- How are test smells treated in the wild? A tale of two empirical studies
- On the diffusion of test smells in automatically generated test code: an empirical study
- On the interplay between software testing and evolution and its effect on program comprehension
- Refactoring Test Code
- Scented since the beginning: On the diffuseness of test smells in automatically generated test code
- Test Smell Detection Tools: A Systematic Mapping Study
- TestQ: Exploring Structural and Maintenance Characteristics of Unit Test Suites
- xUnit test patterns: Refactoring test code

1.1.3.9 Hooks Everywhere

Definition:

- where the production code has awkward backdoors in it to allow test-time rewiring or intercepting

Also Known As:

- Testing Causes an Abstraction Virus

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.1.3.10 Indecent Exposure

Definition:

- Classes and methods should not expose their internals unless there's a good reason to do so. In test automation code bases, we explicitly separate our tests from our framework. It's important to stay true to this by hiding internal framework code from our test layer. It's a code smell to expose class fields such as web elements used in Page Object classes, or class methods that return web elements. Doing so enables test code to access these DOM-specific items, which is not its responsibility.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Writing good gherkin

1.1.3.11 Interface Sensitivity

Definition:

- Interface Sensitivity is when a test fails to compile or run because some part of the interface of the SUT that is uses has changed.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Categorising Test Smells
- xUnit test patterns: Refactoring test code

1.1.3.12 Mixing Production And Test Code

Definition:

- Unit testing code should not really be deployed to production environments. Therefore, when packaging the code ready for release, the testing code must be somehow excluded, which can be complex depending upon the naming conventions used

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- JUnit Anti-patterns

1.1.3.13 Multiple Points Of Failure

Definition:

- Adding assertions within your framework code is a code smell. It is not this code's responsibility to determine the fate of a test. By doing so, it limits the reusability of the framework code for both negative and positive scenarios.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Writing good gherkin

1.1.3.14 Overly Elaborate Test Code

Definition:

- When the test code duplicates the same logic as the code under test, logic which turns out to be incorrect.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Hacker News on: Software Testing Anti-patterns

1.1.3.15 Overspecification

Definition:

- Tests increasingly serve multiple roles in today's projects. They help us design APIs through test-driven development. They provide confidence that new changes aren't breaking existing functionality. They offer an executable specification of the application. But can we ever get to a point where we have too much testing?

Code Example:

```ruby
require File.dirname(__FILE__) + '/../test_helper'

class ProductsControllerTest < ActionController::TestCase
  def test_something
    product = Product.create(:name => "Frisbee", :price => 5.00)
    get :show, :id => product.id
    assert_response :success
    product = assigns(:product)
    assert_not_nil product
    assert product.valid?
    assert product.name == "Frisbee"
    assert product.price == 5.00
  end
end
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Testing anti-patterns: How to fail with 100% test coverage

1.1.3.16 Overspecified Software

Definition:

- The test focuses on the structure and the expected behaviour of the software rather than on what it should accomplish. Therefore, the software must be designed in a specific way in order that the test can pass.

Also Known As:

- Overcoupled Test

References:

Quality attributes

- - Code Example
- - Cause and Effect

- - Frequency
- - Refactoring

- Categorising Test Smells
- xUnit test patterns: Refactoring test code

1.1.3.17 Overspecified Tests

Definition:

- Tests that specify too many things that aren't genuinely related to the scenario being tested.

Code Example:

```java
@Mock private DataSource dataSource;
@Mock private Mock connection;
@Mock private Mock statement;
@Mock private ResultSet resultSet;
@Test
public void test() throws Exception {
  MockitoAnnotations.initMocks(this);
  systemUnderTest = new OracleDAOImpl();
  systemUnderTest.setDBConnectionManager(connectionManager);
  Set<NACustomerDTO> set = new HashSet<NACustomerDTO>();
  when(connectionManager.getDataSource()).thenReturn(dataSource);
  when(dataSource.getConnection()).thenReturn(connection);
  when(connection.createStatement()).thenReturn(statement);
  when(statement.executeQuery(anyString())).thenReturn(resultSet);
  when(resultSet.next()).thenReturn(false);
  when(resultSet.getLong(1)).thenReturn(1L);
  when(resultSet.getString(2)).thenReturn("7178");
  doNothing().when(resultSet).close();
  stub(systemUnderTest.getNACustomers()).toReturn(set);
  final Set<NACustomerDTO> result = systemUnderTest.getNACustomers();
  verify(connectionManager).getDataSource();
  verify(dataSource).getConnection();
  verify(connection).createStatement();
  verify(statement).executeQuery(anyString());
  verify(resultSet).next();
  verify(resultSet).getLong(1);
  verify(resultSet).getString(2);
  assertNotNull(result);
  verify(connectionManager).getDataSource().getConnection();
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency

- - Refactoring

- Bad tests, good tests

1.1.3.18 Plate Spinning

Definition:

- A test that is at risk of exiting prematurely because it does not properly wait for the results of external calls.

Code Example:

```
// Subject under test
function download (path, cb) {
  get(path, function (er, value) {
    if (er) return cb(er)
    insert(value.id, value, function (er) {
      cb(er)
    })
  })
}

// Test
module.exports = {
  beforeEach: function () {
    post('/interests', {id: 42, interests: ['Ruby on Rails']}, function () {})
  },
  getsAndThenInserts: function (done) {
    download('/interests', function (er) {
      select(42, function (er, value) {
        assert.deepEqual(value, {id: 42, interests: ['Ruby on Rails']})
        done(er)
      })
    })
  }
}

// Fake production implementations to simplify example test of subject
var _ = require('lodash')
var resources = {}
function post (path, value, cb) {
  randomTimeout(function () {
    resources[path] = value
    cb(null)
  }, 10, 30)
}

function get (path, cb) {
  randomTimeout(function () {
    cb(null, resources[path])
  }, 15, 50)
}
```

(continues on next page)

(continued from previous page)

```
var db = {}
function insert (id, value, cb) {
  randomTimeout(function () {
    db[id] = value
    cb(null)
  })
}

function select (id, cb) {
  randomTimeout(function () {
    cb(null, db[id])
  })
}

function randomTimeout (func, low, high) {
  setTimeout(func, _(low || 10).range(high || 100).sample())
}

// Exclude this test from CI, since it's erratic
if (process.env.CI) module.exports = {}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- A workbook repository of example test smells and what to do about them

1.1.3.19 Production Logic In Test

Definition:

- Some forms of Conditional Test Logic are found in the result verification section of our tests.

Code Example:

```
public void testCombinationsOfInputValues() {
   // Set up fixture
   Calculator sut = new Calculator();

   int expected; // TBD inside loops

   for (int i = 0; i < 10; i++) {
     for (int j = 0; j < 10; j++) {
       // Exercise SUT
       int actual = sut.calculate( i, j );
```

(continues on next page)

(continued from previous page)

```
      // Verify result
      if (i==3 & j==4) // special case
        expected = 8;
      else
        expected = i+j;

      assertEquals(message(i,j), expected, actual);
    }
  }
}

private String message(int i, int j) {
  return "Cell( " + String.valueOf(i)+ ","
  + String.valueOf(j) + ")";
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- xUnit test patterns: Refactoring test code

1.1.3.20 Test Dependency In Production

Definition:

- Production executables depend on test executables.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- xUnit test patterns: Refactoring test code

1.1.3.21 Test Hook

Definition:

- Conditional logic within the SUT determines whether the "real" code or test-specific logic is run.

Code Example:

```
class ApplicationController < ActionController::Base
  unless Rails.env.test?
    before_filter :require_login
  end
end
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Rails Testing Antipatterns
- xUnit test patterns: Refactoring test code

1.1.3.22 Test Logic In Production Code

Definition:

- The code that is put into production contains logic that should be exercised only during tests.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Automatic generation of smell-free unit tests
- Enhancing developers' awareness on test suites' quality with test smell summaries
- Inspecting Automated Test Code: A Preliminary Study
- xUnit test patterns: Refactoring test code

1.1.3.23 Test Tautology

Definition:

- Generally speaking one does not want to duplicate the logic between tests and code. So replicating a regexp or something else in the test is not an option.

Code Example:

```
Assertions.assertThat(processTemplate("param1", "param2")).isEqualTo(String.format("this␣
↪is '%s', and this is '%s'", param1, param2));
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- How to write good tests

1.1.3.24 Tests Cluttered With Business Logic

Definition:

- Tests that mix details of system business logic with steps of test scenario are hard to read and maintain.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-patterns in test automation

1.1.3.25 Tests Require Too Much Intimate Knowledge Of The Code To Run

Definition:

- Tests only need to know about the methods they are testing, and even then only the interface or what is going in and coming out of them not specific implementation details. This particular anti-pattern or set of them comes from attempts to get 100% code coverage.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-Patterns In Unit Testing

1.1.3.26 The Telltale Heart

Definition:

- where the production code is repeatedly calculating and returning values that are only used at test time.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.1.3.27 The Ugly Mirror

Definition:

- When you occasionally find yourself staring at a spec that looks exactly like the code under test, there's surprisingly little win left to enjoy.

Also Known As:

- Tautological tests

Code Example:

```ruby
require "test/unit"

User = Struct.new(:first_name, :last_name, :email) do
  def to_s
    "#{last_name}, #{first_name} <#{email}>"
  end
end

class UserTest < Test::Unit::TestCase
  def test_to_s_includes_name_and_email
    user = User.new("John", "Smith", "jsmith@example.com")
    assert_equal "#{user.last_name}, #{user.first_name} <#{user.email}>", user.to_s
  end
end
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-Patterns - Digital Tapestry
- Java: code duplication in classes and their junit test cases
- Smells in software test code: A survey of knowledge in industry and academia
- Testing anti-patterns: How to fail with 100% test coverage
- Testing anti-patterns: The ugly mirror

1.1.3.28 Trying To Test The Untestable

Definition:

- When you try to write tests for complex business functionality without refactoring the source.

Code Example:

- No code examples yet…

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Testing the Untestable and Other Anti-Patterns

1.1.3.29 Ui Tests Should Not Expose Interactions With Web Elements

Definition:

- The benefit of using Page Objects is that they abstract implementation logic from the tests. The tests can be focused on the scenarios and not implementation. The idea is that the scenario doesn't change, but the implementation does.

Code Example:

- No code examples yet…

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Top 17 Automated Testing Best Practices (Supported By Data)

1.1.3.30 Well, My Setup Works

Definition:

- a test does not share enough of the setup code used in production, so the test setup can deviate from the production code meaningfully, or at best duplicates production code unnecessarily

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.1.4 Mock and stub related

1.1.4.1 Excessive Mocking

Definition:

- Refers to a test case which requires many mocks to run [35, 45]. The system under test may need a complex environment, specific resources, or depend on 3rd party systems which cannot be provided during test execution. Running tests against a database is a canonical example. If the database isn't available, all tests will fail even though the system under test might be completely bug-free. A common solution is to mock those potential failures by mimicking the behavior of real objects. Developers also use mocks when testing authentication, but sometimes struggle with strategy.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency

- - Refactoring

- Did You Remember To Test Your Tokens?
- Unit Testing Smells: What Are Your Tests Telling You?

1.1.4.2 Is Mockito Working Fine?

Definition:

- When the mock framerwork is tested intead of the SUT

Code Example:

```
@Test
public void testFormUpdate() {
  // given
  Form f = Mockito.mock(Form.class);
  Mockito.when(f.isUpdateAllowed()).thenReturn(true);
  // when
  boolean result = f.isUpdateAllowed();
  // then
  assertTrue(result);
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Bad tests, good tests

1.1.4.3 Making A Mockery Of Design

Definition:

- where pure functions have to be dependency injected so they can be mocked.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.1.4.4 Mismatched Stubs

Definition:

- Stubbed methods called with arguments that differ from those specified for the stub

Code Example:

- No code examples yet…

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Use of test doubles in Android testing: an in-depth investigation

1.1.4.5 Mock Everything

Definition:

- If everything is mocked, are we really testing the production code? Don't hesitate to not mock!

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- How to write good tests

1.1.4.6 Mock Happy

Definition:

- If one uses too many mock objects that interact with each other, the test will only test the interaction between them instead of the functionality of the production code.

Also Known As:

- The Mockery

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Categorising Test Smells
- Smells in software test code: A survey of knowledge in industry and academia
- Smells of Testing (signs your tests are bad)

1.1.4.7 Mock Madness

Definition:

- where even near-primitive values like POJOs are being mocked, just because.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.1.4.8 Mock'Em All

Definition:

- When a test overuse all kinds of test double, even when it is not really the best option

Code Example:

```
@Test
public void shouldAddTimeZoneToModelAndView() {
  //given
  Context context = mock(Context.class);
  ModelAndView modelAndView = mock(ModelAndView.class);
  given(context.getTimezone()).willReturn("timezone X");
  //when
  new UserDataInterceptor(context)
  .postHandle(null, null, null, modelAndView);
  //then
```

(continues on next page)

(continued from previous page)

```
  verify(modelAndView).addObject("timezone", "timezone X");
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Bad tests, good tests

1.1.4.9 Mockers Without Borders

Definition:

- Tests demonstrate little rhyme or reason for whether a given test ought to fake a particular dependency or call through to the real thing.

Code Example:

```
# Subject under test
class App
  def pay_merchants(start_date, end_date)
    transactions = fetch(start_date, end_date)
    purchase_orders = create_purchase_orders(group_by_merchant(transactions))
    submit(purchase_orders)
  end
end

# Test
class MockersWithoutBorders < SmellTest
  def setup
    @subject = App.new
    super
  end

  def test_pays_merchants_with_totals
    transactions = [
      { merchant: 'Nike', desc: 'Shoes', amount: 119.20 },
      { merchant: 'Nike', desc: 'Waterproof spray', amount: 10.10 },
      { merchant: 'Apple', desc: 'iPad', amount: 799.99 },
      { merchant: 'Apple', desc: 'iPad Cover', amount: 59.99 }
    ]
    start_date = Date.civil(2015, 1, 1)
    end_date = Date.civil(2015, 12, 31)
    stub(@subject, :fetch, transactions, [start_date, end_date])
    verify(@subject, :submit, [
```

(continues on next page)

(continued from previous page)

```
      { merchant: 'Nike', total: 129.30 },
      { merchant: 'Apple', total: 859.98 }
    ])

    @subject.pay_merchants(start_date, end_date)
  end
end

# Fake production implementations to simplify example test of subject
class App
  def fetch(start_date, end_date)
    # Imagine something that hits a data store here
  end

  def group_by_merchant(transactions)
    transactions.group_by { |t| t[:merchant] }
  end

  def create_purchase_orders(transactions_by_merchant)
    transactions_by_merchant.map { |(merchant, transactions)|
      {
        merchant: merchant,
        total: transactions.map {|h| h[:amount] }.reduce(:+)
      }
    }
  end

  def submit(purchase_orders)
    # Imagine something that hits a payment processor here
  end
end
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- A workbook repository of example test smells and what to do about them

1.1.4.10 Mocking A Mocking Framework

Definition:

- Occurrs when the test case tests the applied mocking framework

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Did You Remember To Test Your Tokens?

1.1.4.11 Mocking What You Don'T Own

Definition:

- When mocking an external API the test cannot be used to drive the design, the API belongs to someone else ; this third party can and will change the signature and behaviour of the API.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-patterns of automated testing
- How to write good tests

1.1.4.12 Mockito Any() Vs. Isa()

Definition:

- Misuse of mockito's matchers classes to type checks

Code Example:

```
verify(async).execute(any(AddOrganizationAction.class),
  any(AsyncCallback.class));

// wrong pass
verify(async).execute(any(AddPersonToOrganizationAction.class),
  any(AsyncCallback.class));
```

```
verify(async).execute(isA(AddOrganizationAction.class),
  any(AsyncCallback.class));
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Bad tests, good tests

1.1.4.13 Overmocking

Definition:

- where tests are testing situations that are guaranteed to pass as they're whitebox tested against perfect mocks that do not indicate anything to do with reality.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.1.4.14 Overuse Mocking

Definition:

- The problem is using too many mocks. Although mocks are inevitable in cases, you should be aware of the dark side of mocking. It could cause a false negative case, and the defect can only be found when it's too late.

Code Example:

- No code examples yet…

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency

- - Refactoring

- Unveiling 6 Anti-Patterns in React Test Code: Pitfalls to Avoid

1.1.4.15 Remote Control Mocking

Definition:

- where a class that depends on a service is tested with those service's complex dependencies mocked, rather than the service itself being mocked.

Code Example:

```
// pseudocode

given: mock connection factory will return a mock connection
given: mock connection, when queried will return a test collection
given: repository with mock connection factory
given: controller with repository
when: controller handles a get request
then: the result will be the test collection as json
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.1.4.16 Subclass To Test

Definition:

- Need to fake, mock or stub methods in a ClassUnderTest, and/or classes it uses.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Subclass To Test Anti Pattern

1.1.4.17 Surreal

Definition:

- A test whose use of test doubles is so confusing, it's hard to tell what the test is even doing at run-time.

Code Example:

```ruby
# Subject under test
def weigh_clothes(clothes)
  clothes.map { |item| item.weight }.reduce(:+).round
end

# Test
class Surreal < SmellTest
  def setup
    stub(FACTORS, :size, 9, ["S"])
    super
  end

  def test_adds_weights
    small_wet_sock = Clothing.new("S", "sock", "wet")
    large_dry_jacket = Clothing.new("L", "jacket", "dry")
    stub(large_dry_jacket, :weight, 8)
    xl_soaked_pants = OpenStruct.new(weight: 15)

    result = weigh_clothes([small_wet_sock, large_dry_jacket, xl_soaked_pants])

    assert_equal 26, result
  end
end

# Fake production implementations to simplify example test of subject
class Clothing
  def initialize(size, type, wetness)
    @size = size
    @type = type
    @wetness = wetness
  end

  def weight
    return 1 *
      FACTORS.size(@size) *
      FACTORS.type(@type) *
      FACTORS.wetness(@wetness)
  end
end

class Factors
  def size(size)
    case size
    when "S" then 0.75
    when "M" then 1
    when "L" then 1.25
```

(continues on next page)

(continued from previous page)

```
    when "XL" then 1.5
    else 1
    end
  end

  def type(type)
    case type
    when "sock" then 0.2
    when "shirt" then 1
    when "pants" then 2
    when "jacket" then 3
    else 1
    end
  end

  def wetness(wetness)
    case wetness
    when "dry" then 1
    when "moist" then 1.1
    when "wet" then 1.6
    when "soaked" then 2.5
    else 1
    end
  end
end
FACTORS = Factors.new
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- A workbook repository of example test smells and what to do about them

1.1.4.18 Testing The Framework

Definition:

- If you're using a framework you pretty much have to assume it works. Testing it along with your other tests only clutters things up.

References:

Quality attributes

- - Code Example

- - Cause and Effect
- - Frequency
- - Refactoring

- Smells of Testing (signs your tests are bad)

1.1.4.19 The Dead Tree

Definition:

- A test which where a stub was created, but the test wasn't actually written.

Also Known As:

- Process Compliance Backdoor

Code Example:

```
class TD_SomeClass {
  public void testAdd() {
    assertEquals(1+1, 2);
  }
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Categorising Test Smells
- Unit testing Anti-patterns catalogue

1.1.4.20 The Mockery

Definition:

- Sometimes mocking can be good, and handy. But sometimes developers can lose themselves and in their effort to mock out what isn't being tested. In this case, a unit test contains so many mocks, stubs, and/or fakes that the system under test isn't even being tested at all, instead data returned from mocks is what is being tested.

Also Known As:

- Mock Happy, Mock-Overkill

References:

Quality attributes

- - Code Example

- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-Patterns - Digital Tapestry
- Test-Driven Development: TDD Anti-Patterns
- Unit Testing Anti-Patterns, Full List
- Unit testing Anti-patterns catalogue

1.1.4.21 Unnecessary Stubs

Definition:

- Stubbed method never called during test execution

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Use of test doubles in Android testing: an in-depth investigation

1.1.5 Violating coding best practices

1.1.5.1 Accidental Test Framework

Definition:

- related to over exertion asserts, where there's an ad-hoc bit of what should be a test library, this also includes home-made shallow implementations for deep problems like managing resources such as database or file. It also includes manually pumping framework primitives, rather than using the framework as a whole.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.1.5.2 Ambiguous Tests

Definition:

- The test is underspecified and leaves room for interpretation

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Hunting for smells in natural language tests

1.1.5.3 Anonymous Test Case

Definition:

- where we cannot tell from the test name what is being tested

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.1.5.4 Anonymous Test

Definition:

- An anonymous test is a test whose name is meaningless as it doesn't express the purpose of the test in the current context. However tests can be regarded as documentation, and the name is an important part of that as it should abstract what the test is all about.

Also Known As:

- Unclear Naming, Naming Convention Violation

Code Example:

```
@Test
public void test1() {
    // test user login
```

(continues on next page)

(continued from previous page)

```
    LoginPage.login("user", "password");
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Assessing test quality - TestLint
- Automatic generation of smell-free unit tests
- Categorising Test Smells
- Detection of test smells with basic language analysis methods and its evaluation
- Generated Tests in the Context of Maintenance Tasks: A Series of Empirical Studies
- Rule-based Assessment of Test Quality
- Test Smell Detection Tools: A Systematic Mapping Study
- Test Smells - The Coding Craftsman

1.1.5.5 Bad Documentation Comment

Definition:

- A documentation comment does not conform to its format. Documentation comments like T3Doc need to conform to certain formatting rules and appear at certain positions in the source code.

Code Example:

```
/**
* This function does something.
* @param imParam a very important parameter
*/
function exampleFunction(integer inParam) {
  // ...
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- An approach to quality engineering of TTCN-3 test specifications
- Pattern-based Smell Detection in TTCN-3 Test Suites
- Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites

1.1.5.6 Bad Naming

Definition:

- When a test fails, or when the test base requires maintenance, the test names are the first thing developers will generally attempt to understand before they apply changes to the test or the code being tested. If test names are poor quality, developers will need to spend time reading the code and determining how the test's actual behavior is related to its name

Code Example:

```
function calculateSomething() {
  // ...
}

function calculateSomethingElse() {
  // ...
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- An approach to quality engineering of TTCN-3 test specifications
- Improving Student Testing Practices through a Lightweight Checklist Intervention.
- Pattern-based Smell Detection in TTCN-3 Test Suites
- Test Naming Failures. An Exploratory Study of Bad Naming Practices in Test Code
- Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites
- What We Know About Smells in Software Test Code

1.1.5.7 Badly Structured Test Suite

Definition:

- The structure of the test suite does not follow the structure of the tested functionality

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Hunting for smells in natural language tests

1.1.5.8 Blethery Prefixes

Definition:

- Where the implementation of a single line of test code is prefixed by a lot of characters before we get to the point

Code Example:

```java
@Test
void someTest() {
    Mockito.when(someMock.get())
    .thenReturn(123);

    ...
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.1.5.9 Comments Only Test

Definition:

- A test that has been put into comments

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Rule-based Assessment of Test Quality
- Test Smell Detection Tools: A Systematic Mapping Study

1.1.5.10 Constant Actual Parameter Value

Definition:

- The value of an actual parameter is the same for all occurances. In contrast to Unused Parameter (4.3.1), the parameter is in use within the declaring entity and must not simply be removed. The declaring entity could be a template or a behavioral entity (function, test case or altstep).

Code Example:

```
template myType t (charstring p1, integer p2) := {
  field1 := true,
  field2 := p2,
  field3 := p1
}

function f() runs on myComponent {
  // ...
  p.send(templateA("foo", 42));
  // ...
  p.send(templateA("foo", 42));
  // ...
  p.send(templateA("foo", 43));
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- An approach to quality engineering of TTCN-3 test specifications
- Pattern-based Smell Detection in TTCN-3 Test Suites
- Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites

1.1.5.11 Dead Field

Definition:

- Occurs when a class or its super classes have fields that are never used by any test method. Often dead fields are inherited. This can indicate a non-optimal inheritance structure, or that the super class conflicts with the single responsibility principle. Also, dead fields within the test class itself can indicate incomplete or deprecated development activities.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Automated Detection of Test Fixture Strategies and Smells
- Automatic generation of smell-free unit tests
- Strategies for avoiding text fixture smells during software evolution
- Test Smell Detection Tools: A Systematic Mapping Study

1.1.5.12 Default Test

Definition:

- By default Android Studio creates default test classes when a project is created. These template test classes are meant to serve as an example for developers when writing unit tests and should either be removed or renamed. Having such files in the project will cause developers to start adding test methods into these files, making the default test class a container of all test cases and violate good testing practices. Problems would also arise when the classes need to be renamed in the future.

Code Example:

```
public class ExampleUnitTest {
        @Test
        public void addition_isCorrect() throws Exception {
                assertEquals(4, 2 + 2);
        }

        @Test
        public void shareProblem() throws InterruptedException {
                .....
```

(continues on next page)

(continued from previous page)

```
                Observable.just(200)
                .subscribeOn(Schedulers.newThread())
                .subscribe(begin.asAction());
                begin.set(200);
                Thread.sleep(1000);
                assertEquals(beginTime.get(), "200");
                .....
        }
        .....
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- An Exploratory Study on the Refactoring of Unit Test Files in Android Applications
- Automatic Identification of High-Impact Bug Report by Product and Test Code Quality
- Automatic generation of smell-free unit tests
- Handling Test Smells in Python: Results from a Mixed-Method Study
- On the distribution of test smells in open source Android applications: an exploratory study
- On the influence of Test Smells on Test Coverage
- On the test smells detection: an empirical study on the jnose test accuracy
- On the use of test smells for prediction of flaky tests
- PyNose: A Test Smell Detector For Python
- Software Unit Test Smells
- Test Smell Detection Tools: A Systematic Mapping Study
- What the Smell? An Empirical Investigation on the Distribution and Severity of Test Smells in Open Source Android Applications
- tsDetect: an open source test smells detection tool

1.1.5.13 Directly Executing Javascript

Definition:

- Since WebDriver can directly execute any arbitrary JavaScript, it can be tempting to bypass DOM manipulation and just run the JavaScript.

Code Example:

```
public void RemoveTea(string teaType)
{
  (driver as IJavaScriptExecutor).ExecuteScript(string.Format(&quot;viewModel.tea.types.
↪removeTeaType(\&quot;{0}\&quot;);&quot;, teaType));
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Five automated acceptance test anti-patterns

1.1.5.14 Disabled Test

Definition:

- Disabling failing tests temporarily can be seen as an added flexibility for developers to alleviate maintenance difficulties, but it may introduce technical debt.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- How disabled tests manifest in test maintainability challenges?

1.1.5.15 Empty Method Category

Definition:

- A test method with an empty method category

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Rule-based Assessment of Test Quality
- Test Smell Detection Tools: A Systematic Mapping Study

1.1.5.16 Empty Test-Method Category

Definition:

- A test method with an empty test method category

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Rule-based Assessment of Test Quality
- Test Smell Detection Tools: A Systematic Mapping Study

1.1.5.17 Empty Test

Definition:

- Occurs when a test method has no executable statements. Such methods are possibly created for debugging purposes without being deleted or contain commented-out test statements. An empty test method can be considered problematic and more dangerous than not having a test case at all since JUnit will indicate that the test passes even if there are no executable statements present in the method body. As such, developers introducing behavior-breaking changes into production class, will not be notified of the alternated outcomes as JUnit will report the test as passing.

Code Example:

```
public void testCredGetFullSampleV1() throws Throwable{
//        ScrapedCredentials credentials =  innerCredTest(FULL_SAMPLE_v1);
//        assertEquals("p4ssw0rd", credentials.pass);
//        assertEquals("user@example.com",credentials.user);
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- A survey on test practitioners' awareness of test smells
- An Exploratory Study on the Refactoring of Unit Test Files in Android Applications
- Automatic Identification of High-Impact Bug Report by Product and Test Code Quality
- Automatic generation of smell-free unit tests
- Developers perception on the severity of test smells: an empirical study
- Handling Test Smells in Python: Results from a Mixed-Method Study
- How are test smells treated in the wild? A tale of two empirical studies
- Investigating Severity Thresholds for Test Smells
- Investigating Test Smells in JavaScript Test Code
- Machine Learning-Based Test Smell Detection
- On the Distribution of "Simple Stupid Bugs" in Unit Test Files: An Exploratory Study
- On the diffusion of test smells and their relationship with test code quality of Java projects
- On the distribution of test smells in open source Android applications: an exploratory study
- On the influence of Test Smells on Test Coverage
- On the test smells detection: an empirical study on the jnose test accuracy
- On the use of test smells for prediction of flaky tests
- PyNose: A Test Smell Detector For Python
- Software Unit Test Smells
- Test Smell Detection Tools: A Systematic Mapping Study
- TestQ: Exploring Structural and Maintenance Characteristics of Unit Test Suites
- The secret life of test smells-an empirical study on test smell evolution and maintenance
- Understanding Testability and Test Smells
- What the Smell? An Empirical Investigation on the Distribution and Severity of Test Smells in Open Source Android Applications
- tsDetect: an open source test smells detection tool

1.1.5.18 Erratic Test

Definition:

- One or more tests are behaving erratically; sometimes they pass and sometimes they fail.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Automatic generation of smell-free unit tests
- Inspecting Automated Test Code: A Preliminary Study
- Rails Testing Antipatterns
- xUnit test patterns: Refactoring test code

1.1.5.19 Erratic Tests

Definition:

- Tests that will pass or fail without you changing anything

Code Example:

```
first(".active").click

all(".active").each(&:click)

execute_script("$('.active').focus()")

expect(find_field("Username").value).to eq("Joe")

expect(find(".user")["data-name"]).to eq("Joe")

expect(has_css?(".active")).to eq(false)
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Rails Testing Antipatterns

1.1.5.20 Everything Is A Property

Definition:

- Where a test class keeps what should be temporary variables in instance variables

Also Known As:

- Temporary Test Property

Code Example:

```
class ParagraphAnalyzerTest {
    private String analyzed;
    private ParagraphAnalyzer analyzer = new ParagraphAnalyzer();

    @Test
    void nouns() {
        analyzed = analyzer.justNouns("This is a word");
        assertThat(analyzed).isEqualTo("word");
    }

    @Test
    void verbs() {
        analyzed = analyzer.justVerbs("This is a word");
        assertThat(analyzed).isEqualTo("is");
    }

    @Test
    void ends() {
        analyzed = analyzer.first("This is a word");
        assertThat(analyzed).isEqualTo("This");

        analyzed = analyzer.last("This is a word");
        assertThat(analyzed).isEqualTo("words");
    }
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.1.5.21 Flaky Locator

Definition:

- A key component to making UI automation work is to provide your tool with identifiers to the elements that you'd like it to find and interact with. Using flaky locators—ones that are not durable—is an awful code smell.

Code Example:

```
/html/body/div/div/div/div/div[2]/label/span[2]
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Writing good gherkin

1.1.5.22 Flaky Test

Definition:

- A flaky test is a test that could fail or pass for the same configuration. Flaky tests could be harmful for developers because their failures do not always indicate bugs in the code.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- JTDog: a Gradle Plugin for Dynamic Test Smell Detection
- Revisiting Test Impact Analysis in Continuous Testing From the Perspective of Code Dependencies
- Static Test Flakiness Prediction
- Static Test Flakiness Prediction
- Static test flakiness prediction: How Far Can We Go?
- Surveying the developer experience of flaky tests
- What We Know About Smells in Software Test Code

1.1.5.23 Fully-Parameterized Template

Definition:

- All fields of a template are defined by formal parameters.

Code Example:

```
type record MyMessageType {
    integer field1,
    charstring field2,
    boolean field3
}

template MyMessageType exampleTemplate(integer i, charstring c, boolean b) := {
    field1 := i,
    field2 := c,
    field3 := b
}

function f() runs on MyComponent {
    // ...
    p.send(exampleTemplate(42, "dent", true));
    // ...
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- An approach to quality engineering of TTCN-3 test specifications
- Pattern-based Smell Detection in TTCN-3 Test Suites
- Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites

1.1.5.24 Goto Statement

Definition:

- A goto statement is used. The use of goto statements is inadvisable and should be avoided.

Code Example:

```
function f ( integer i ) runs on ExampleComponent {
  var integer MyVar := i ;
  label L1 ;
  MyVar := 2  MyVar ;
  if (MyVar < 2000) {
```

(continues on next page)

(continued from previous page)

```
    goto L1 ;
  }
  MyVar2 := f2(MyVar);
  if (MyVar2 > MyVar) {
    goto L2 ;
  }
  p.send(MyVar);
  p.receive -> value MyVar2;
  label L2 ;
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- An approach to quality engineering of TTCN-3 test specifications
- Pattern-based Smell Detection in TTCN-3 Test Suites
- Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites

1.1.5.25 Having Flaky Or Slow Tests

Definition:

- A test that sometimes fails and sometimes passes (without any code changes in between) is unreliable and undermines the whole testing suite

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Software Testing Anti-patterns

1.1.5.26 Hidden Meaning

Definition:

- Where something that should be part of the execution of the test, and appear in a test report, is hidden in a comment – essentially comment instead of name

Code Example:

```
@Test
public void restApi() {
    int response = client.get("/endpoint");

    // the status code returned from the get
    // should be OK, indicating
    // the endpoint is healthy
    assertEquals(200, response);
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.1.5.27 Hidden Test Call

Definition:

- Tests calling other tests

Code Example:

```
[TestFixture]
public class HiddenTestCall {
  private LogAnalyzer logan;

  [Test]
  public void CreateAnalyzer_GoodNameAndBadNameUsage() {
    logan.new LogAnalyzer();
    logan.Initialize();
    bool valid = logan.IsValid("abc");
    Assert.That(valid, Is.False);
    CreateAnalyzer_GoodFileName_ReturnsTrue();
  }

  [Test]
  public void CreateAnalyzer_GoodFileName_ReturnsTrue() {
```

(continues on next page)

(continued from previous page)

```
    bool valid = logan.IsValid("abcdefg");
    Assert.That(valid. Is.True);
  }
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Chapter 8. The pillars of good unit tests

1.1.5.28 Ignored Test

Definition:

- JUnit 4 provides developers with the ability to suppress test methods from running. However, these ignored test methods result in overhead since they add unnecessary overhead with regards to compilation time, and increases code complexity and comprehension.

Code Example:

```
@Ignore("disabled for now as this test is too flaky")
public void peerPriority() throws Exception {
  final List addresses = Lists.newArrayList(
      new InetSocketAddress("localhost", 2000),
        new InetSocketAddress("localhost", 2001),
        new InetSocketAddress("localhost", 2002)
  );
    peerGroup.addConnectedEventListener(connectedListener);
    .....
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- A preliminary evaluation on the relationship among architectural and test smells
- An Exploratory Study on the Refactoring of Unit Test Files in Android Applications

- Automatic Identification of High-Impact Bug Report by Product and Test Code Quality
- Automatic generation of smell-free unit tests
- Handling Test Smells in Python: Results from a Mixed-Method Study
- Investigating Severity Thresholds for Test Smells
- Investigating Test Smells in JavaScript Test Code
- On the Distribution of “Simple Stupid Bugs” in Unit Test Files: An Exploratory Study
- On the diffusion of test smells and their relationship with test code quality of Java projects
- On the distribution of test smells in open source Android applications: an exploratory study
- On the influence of Test Smells on Test Coverage
- On the test smells detection: an empirical study on the jnose test accuracy
- On the use of test smells for prediction of flaky tests
- PyNose: A Test Smell Detector For Python
- Pytest-Smell: a smell detection tool for Python unit tests
- Software Unit Test Smells
- Test Smell Detection Tools: A Systematic Mapping Study
- The secret life of test smells-an empirical study on test smell evolution and maintenance
- Understanding Testability and Test Smells
- What the Smell? An Empirical Investigation on the Distribution and Severity of Test Smells in Open Source Android Applications
- tsDetect: an open source test smells detection tool

1.1.5.29 Improper Test Method Location

Definition:

- Smalltalk-developers normally organize their source-code in packages, classcategories and method-categories. Although there is no common convention for doing so, most projects we encountered during our case study are organized in a very similar fashion. However we also noticed that tests are sometimes excluded from that.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Assessing test quality - TestLint

1.1.5.30 Inconsistent Hierarchy

Definition:

- The macro component hierarchy is inconsistent with the structure of the GUI

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Bad smells and refactoring methods for GUI test script

1.1.5.31 Integration Test, Masquerading As Unit Test

Definition:

- where there are too many layers involved in making a unit test, so it runs too long

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.1.5.32 Intermittent Test Failures

Definition:

- You run them with the same inputs over and over again, and usually, but not always, they produce the same output.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Unit Testing Smells: What Are Your Tests Telling You?

1.1.5.33 Is There Anybody There?

Definition:

- Any flickering test that occasionally breaks a build – bad test or bad code?

Code Example:

```
@Test
public void testMethod() {
    int expected = 5;
    int actual = MyUnstableClass.doSomethingWithRandomReturns();
    assertEquals(expected, actual);
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.1.5.34 Lack Of Encapsulation

Definition:

- The implementation details of a test are not properly hidden in the implementation layer and start appearing in its acceptance criteria.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Bad smells and refactoring methods for GUI test script
- On the Maintenance of System User Interactive Tests
- Smells in System User Interactive Tests

1.1.5.35 Lack Of Macro Events

Definition:

- A macro component does not have any macro events.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Bad smells and refactoring methods for GUI test script

1.1.5.36 Literal Pollution

Definition:

- When writing tests for the application code it is mostly required also to provide some data to be able to test the functionality. This is mostly done by defining literals in the test code. However an excessive use of literals can cause severe problems: * Too many literals are distracting and obfuscate the functionality and purpose of a test. This makes a test hard to read and understand. * The same or similar test data is often repeated within a test or testsuite. This is often a consequence of simply extending or adding tests without actually designing them. The result is a test-suite that is extremely hard to maintain and refactor. We detected such Duplication in harvesting our case study.

Code Example:

```
UrlTest >> #testUsernamePasswordPrinting
  #('http://user:pword@someserver.blah:8000/root/index.html'
  'http://user@someserver.blah:8000/root/index.html'
  'http://user:pword@someserver.blah/root/index.html'
  ) do: [ :urlText | self should: [ urlText = urlText asUrl asString ] ].
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Assessing test quality - TestLint

1.1.5.37 Long Parameter List

Definition:

- High number of formal parameters.

Code Example:

```
function f1(integer i1, integer i2, integer i3, integer i4, integer i5, integer i6) {
  // some behavior...
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- A Refactoring Tool for TTCN-3
- An approach to quality engineering of TTCN-3 test specifications
- Bad smells and refactoring methods for GUI test script
- Pattern-based Smell Detection in TTCN-3 Test Suites
- Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites

1.1.5.38 Lost Test

Definition:

- The number of tests being executed in a test suite has dropped (or has not increased as much as expected). We may notice this directly if we are paying attention to test counts or we may find a bug that should have been caused by a test that we know exists but upon poking around we discover that the test has been disabled.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- xUnit test patterns: Refactoring test code

1.1.5.39 Meaningless Variable Names

Definition:

- Unable to tell which variable is for what

Code Example:

- No code examples yet…

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Development Patterns and Anti-Patterns - Medium

1.1.5.40 Messy Test

Definition:

- Tests that contain repeated code, copy&paste, disorganized structure and literal values

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- A testing anti-pattern safari

1.1.5.41 Missed Skip Rotten Green Test

Definition:

- Test methods contain guards to stop their execution early under certain conditions.

Code Example:

```
@Test
    public void testNormalizedKeysGreatSmallAscDescHalfLenght(){
    TypeComparator<T> comparator = getComparator(true);
    if (not(comparator.supportsNormalizedKey())){
        return;
    }
```

(continues on next page)

(continued from previous page)

```
    testNormalizedKeysGreatSmall(true, comparator,true);
    testNormalizedKeysGreatSmall(false, comparator,true);
    }
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Rotten green tests in Java, Pharo and Python

1.1.5.42 Missing Variable Definition

Definition:

- A variable or out parameter is read before its value has been defined. Access to undefined variables might result in unpredictable behavior.

Also Known As:

- Ur Data Flow Anomaly

Code Example:

```
function f (in integer i, out integer j) {
  var integer k := 1, l;
  j := i + j + k + l;
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- An approach to quality engineering of TTCN-3 test specifications
- Pattern-based Smell Detection in TTCN-3 Test Suites
- Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites

1.1.5.43 Missing Verdict

Definition:

- A test case does not set a verdict. Normally a test case should set a verdict before terminating.

Code Example:

```
testcase exampleTestCase ( ) runs on ExampleComponent {
  timer tguard;
  // . . .
  tguard.start(10.0);
  alt {
      [] pt.receive(aMessageOne) {
          tguard.stop;
          setverdict(pass);
          pt.send(aMessageTwo);
      }
      [] anyport.receive {
          repeat;
      }
      [] tguard.timeout {
          stop;
      }
  }
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- An approach to quality engineering of TTCN-3 test specifications
- Pattern-based Smell Detection in TTCN-3 Test Suites
- Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites

1.1.5.44 Mistaken Identity

Definition:

- where the name of a test gives a contrary opinion to the meaning of the test

References:

Quality attributes

- - Code Example
- - Cause and Effect

- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.1.5.45 Mixed Selectors

Definition:

- The problem of mixing up all the methods of a test-class is that it is harder to allocate and differentiate accessors, xtures, utilities and test-methods. By putting each type of method into a different method category, especially strictly separating test-methods from other methods we get a better structure of the test-class. A better and cleaner structure helps in understanding the test-suite, the xtures and all the test-methods.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Assessing test quality - TestLint
- Automatic generation of smell-free unit tests
- Rule-based Assessment of Test Quality
- Test Smell Detection Tools: A Systematic Mapping Study

1.1.5.46 Name-Clashing Import

Definition:

- An imported element causes a name clash with a declaration in the importing module or another imported element.

Code Example:

```
module Foo {
  const charstring MY_CONST := "foo";
}

module Bar {
  import from Foo all;

  const charstring MY_CONST := "bar";

  function f(in charstring s) return boolean {
    if (MY_CONST == s) {
      return true;
```

(continues on next page)

(continued from previous page)

```
    }
    return false;
  }
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Pattern-based Smell Detection in TTCN-3 Test Suites

1.1.5.47 Non-Functional Statement

Definition:

- When there is an empty scope within a test method. In Python, this occurs with the use of the pass keyword within a scope

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Handling Test Smells in Python: Results from a Mixed-Method Study
- TEMPY: Test Smell Detector for Python

1.1.5.48 Nondeterministic Test

Definition:

- Test failures occur at random even when only a single Test Runner is running tests.

Code Example:

```
Suite.run() --> Test 3 fails
Suite.run() --> Test 3 crashes
Suite.run() --> All tests pass
Suite.run() --> Test 3 fails
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Categorising Test Smells
- xUnit test patterns: Refactoring test code

1.1.5.49 Opaque Output

Definition:

- This occurs when the output does not show what was tested.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Design Patterns for Database API Testing 1: Web Service Saving 2 – Code

1.1.5.50 Over-Eager Helper

Definition:

- where there's a helper method that probes the system and then performs an assertion, rather than return its result for the caller to assert.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.1.5.51 Over-Specific Runs On

Definition:

- A behavioral entity (function, test case or altstep) is declared to run on a component, but uses only elements of this component's super-component or no elements of the component at all.

Code Example:

```
type component BaseComponentType {
  port OutPort pOut;
}

type component ExtendedComponentType extends BaseComponentType {
  port InPort pIn;
  timer t;
}

function f(string aMessage) runs on ExtendedComponentType {
  if (checkSomething()) {
    pOut.sent(aMessage);
  }
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- An approach to quality engineering of TTCN-3 test specifications
- Pattern-based Smell Detection in TTCN-3 Test Suites
- Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites

1.1.5.52 Overreferencing

Definition:

- It is about test-methods referencing many times classes from the application code. The main problem with an Overreferencing Test is that it causes a lot of unnecessary dependencies towards the model code. That distracts from the goal of the test.
- Test creating unnecessary dependencies and causing duplication

Code Example:

```
BooleanTypesTest >> #testTrueFalseSubtype
  | system boolType boolMetaType |
  system := TPStructuralTypeSystem new.
```

(continues on next page)

(continued from previous page)

```
boolType := TPClassType on: Boolean.

self assert: (system is: (TPClassType on: True) subtypeOf: boolType).

self assert: (system is: (TPClassType on: False) subtypeOf: boolType).

self assert: (system is: (TPClassType on: False) subtypeOf: (TPClassType on: True)).

self assert: (system is: (TPClassType on: True) subtypeOf: (TPClassType on: False)).

boolMetaType := TPClassType on: Boolean class.

self assert: (system is: (TPClassType on: True class) subtypeOf: boolMetaType).

self assert: (system is: (TPClassType on: False class) subtypeOf: boolMetaType).

self assert: (system is: (TPClassType on: False class) subtypeOf: (TPClassType on:␣
↪True class)).

self assert: (system is: (TPClassType on: True class) subtypeOf: (TPClassType on:␣
↪False class)).
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Assessing test quality - TestLint
- Automatic generation of smell-free unit tests
- Rule-based Assessment of Test Quality
- Test Smell Detection Tools: A Systematic Mapping Study

1.1.5.53 Proper Organization

Definition:

- Violating testing conventions by using bad organization of methods

References:

Quality attributes

- - Code Example
- - Cause and Effect

- - Frequency
- - Refactoring

- Automatic generation of smell-free unit tests
- Rule-based Assessment of Test Quality
- Test Smell Detection Tools: A Systematic Mapping Study

1.1.5.54 Sensitive Locators

Definition:

- The test uses element identification selectors that have long chains to identify an element in the user interface. e.g. complex x-pass or CSS selector for web application.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- On the Maintenance of System User Interactive Tests
- Smells in System User Interactive Tests

1.1.5.55 Short Template

Definition:

- Template definition is very short (in terms of characters or number of fields).

Code Example:

```
template integer exampleTemplate := 1;

testcase exampleTestCase() runs on ExampleComponent {
  // ...
  pt.send(exampleTemplate);
  // ...
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency

- - Refactoring

- An approach to quality engineering of TTCN-3 test specifications
- Pattern-based Smell Detection in TTCN-3 Test Suites
- Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites

1.1.5.56 Singular Component Variable/Constant/Timer Reference

Definition:

- A component variable, constant or timer is referenced by one single function, test case or altstep only, although other behavioral entities run on the component as well.

Code Example:

```
module SingularComponentVCTReference {
  type port ExamplePort message {
    inout charstring;
  }

  type component c {
    timer t;
    port ExamplePort p;
  }

  function f() runs on c {
    p.send("bar");
    p.send("baz");
  }

  testcase tc() runs on c {
    t.start(10.0);
    alt {
      [] p.receive("foo") {
        p.send("bar");
      }
      [] any port.receive {
        // error handling
      }
      [] t.timeout {
        // error handling
      }
    }
  }
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect

- - Frequency
- - Refactoring

- An approach to quality engineering of TTCN-3 test specifications
- Pattern-based Smell Detection in TTCN-3 Test Suites
- Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites

1.1.5.57 Singular Template Reference

Definition:

- A template definition is referenced only once.

Code Example:

```
template MyMessageType exampleTemplate := {
  field1 := omit,
  field2 := "foo",
  field3 := true
}

testcase exampleTestCase() runs on ExampleComponent {
  // ...
  pt.send(exampleTemplate);
  // ...
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- An approach to quality engineering of TTCN-3 test specifications
- Pattern-based Smell Detection in TTCN-3 Test Suites
- Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites

1.1.5.58 Stop In Function

Definition:

- A function contains a stop statement. If possible, functions should not contain any stop statement, because this can prevent the execution of postambles (e.g. code that has to be executed after each test case). Instead, functions should use return values. However, this smell should be classified weak compared to other smells.

Code Example:

```
function f() {
  timer t := 50;
  t.start();

  alt {
    [] p.receive("foo") {
      t.stop;
      setverdict(pass);
    }
    [] t.timeout {
      setverdict(inconc);
      stop;
    }
  }
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- An approach to quality engineering of TTCN-3 test specifications
- Pattern-based Smell Detection in TTCN-3 Test Suites
- Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites

1.1.5.59 Test Body Is Somewhere Else

Definition:

- When the test method calls another method entirely with no other implementation in the test method – often a sign of missing parameterised test

Code Example:

```
@Test
public void hasCorrectFoo() {
  checkFoo();
}
```

(continues on next page)

(continued from previous page)

```
private static checkFoo() {
  Foo foo = service.getFoo();
  assertThat(foo.getBar()).isEqualTo("baz");
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.1.5.60 Test-Class Name

Definition:

- A test that has a class with a meaningless name.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Automatic generation of smell-free unit tests
- Rule-based Assessment of Test Quality
- Test Smell Detection Tools: A Systematic Mapping Study

1.1.5.61 Test-Method Category Name

Definition:

- A test method has a meaningless name

References:

Quality attributes

- - Code Example
- - Cause and Effect

- - Frequency
- - Refactoring

- Rule-based Assessment of Test Quality
- Test Smell Detection Tools: A Systematic Mapping Study

1.1.5.62 Testing For A Specific Bug

Definition:

- Sometimes you need to reproduce a bug so a new test case is created for reproducing that bug. Most times developers are less descriptive in naming these tests calling them something like "testForBugXYZ"

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-Patterns In Unit Testing

1.1.5.63 Testy Testy Test Test

Definition:

- where the name of a test contains the name test, which is hardly surprising, as it's a test, in a test fixture that's probably annotated with an annotation/attribute called test

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.1.5.64 The Conjoined Twins

Definition:

- Tests that people are calling "Unit Tests" but are really integration tests since they are not isolated from dependencies (file configuration, databases, services, other in other words the parts not being tested in your tests that people got lazy and did not isolate) and fail due to dependencies that should have been stubbed or mocked.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Unit Testing Anti-Patterns, Full List
- Unit testing Anti-patterns catalogue

1.1.5.65 The Enumerator

Definition:

- A unit test where each test case method name is only an enumeration, i.e. test1, test2, test3. As a result, the intention of the test case is unclear, and the only way to be sure is to read the test case code and pray for clarity.

Also Known As:

- Test With No Name

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-Patterns - Digital Tapestry
- Categorising Test Smells
- Test-Driven Development: TDD Anti-Patterns
- Unit Testing Anti-Patterns, Full List

1.1.5.66 The Flickering Test

Definition:

- A test which just occasionally fails, and is generally due to race conditions within the test. Typically occurs when testing something that is asynchronous. Possibly a superset of the Wait and See and The Sleeper anti-patterns.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-Patterns - Digital Tapestry
- Unit testing Anti-patterns catalogue

1.1.5.67 The Forty Foot Pole Test

Definition:

- Afraid of getting too close to the class they are trying to test, these tests act at a distance, separated by countless layers of abstraction and thousands of lines of code from the logic they are checking. As such they are extremely brittle, and susceptible to all sorts of side-effects that happen on the epic journey to and from the class of interest.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-Patterns - Digital Tapestry
- Unit Testing Anti-Patterns, Full List
- Unit testing Anti-patterns catalogue

1.1.5.68 The Sleeper

Definition:

- A test that is destined to fail at some specific time and date in the future. This is often caused by incorrect bounds checking when testing code which uses a Date or Calendar object. Sometimes, the test may fail if run at a very specific time of day, such as midnight.

Also Known As:

- Mount Vesuvius

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-Patterns - Digital Tapestry
- Categorising Test Smells
- Unit testing Anti-patterns catalogue

1.1.5.69 The Stepford Fields

Definition:

- Where (too) many of the fields in test contain the same value, making it hard to spot when a calculation is reading a value from the wrong field, because it works on the test data; this also makes load testing near a cache pretty meaningless

Code Example:

```
name:
 title: Mr
 first: Testing
 last: Testing
company: Testing
```

```
expect(person.getFullName())
  .toBe('Mr Testing Testing');
expect(person.getCompanyName())
    .toBe('Testing');

person.company = 'Testing';

expect(person.getCompanyName())
  .toBe('Testing');
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.1.5.70 The Test With No Name

Definition:

- The test that gets added to reproduce a specific bug in the bug tracker and whose author thinks does not warrant a name of its own. Instead of enhancing an existing, lacking test, a new test is created called testForBUG123. Two years later, when that test fails, you may need to first try and find BUG-123 in your bug tracker to figure out the test's intent.

Also Known As:

- The Enumerator

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-Patterns - Digital Tapestry
- Categorising Test Smells
- Unit testing Anti-patterns catalogue

1.1.5.71 Time Bomb Data

Definition:

- when testing with dates or times, and the choice of test data leads either to a test that starts to fail in the future, because the current date has changed to a point where the test data is wrong, or where the full range of relevant date variations is not covered, leading to a bug in the production code on certain days

References:

Quality attributes

- - Code Example
- - Cause and Effect

- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.1.5.72 Time Bombs

Definition:

- Tests that fail due to ever-so-slightly different time values, during certain days of the week or month, or when a long-running time-sensitive test straddles two hours, days, weeks, months, or years and the code can't handle it.

Code Example:

```
# Subject under test
require 'bigdecimal'
require 'bigdecimal/util'

class TimeCard
  attr_reader :start_time, :end_time

  def initialize(hourly_wage)
    @hourly_wage = hourly_wage
  end

  def punch_in(at = nil)
    @start_time = at || Time.new
  end

  def punch_out(at = nil)
    @end_time = at || Time.new
  end

  def wage_owed
    seconds = (@end_time || Time.new) - @start_time
    hours = seconds / (60 * 60).to_d
    bonus = worked_on_weekend? ? 1.5 : 1

    @hourly_wage * hours * bonus
  end
end

# Test
class TimeBombs < SmellTest
  include UnreliableMinitestPlugin

  def setup
    @subject = TimeCard.new(15)
    @now = Time.new
    super
  end

  def test_punch_in_defaults_to_now
```

(continues on next page)

(continued from previous page)

```
    @subject.punch_in

    assert_equal @subject.start_time.to_ms, @now.to_ms
  end

  def test_calculates_wage_owed
    @subject.punch_in(@now)
    @subject.punch_out(@now + (60 * 60 * 24))

    result = @subject.wage_owed

    assert_equal 360, result.to_i
  end
end

# Fake production implementations to simplify example test of subject
class TimeCard
  def worked_on_weekend?
    return @start_time.wday === 0 ||
      @start_time.wday === 6 ||
      @end_time.wday === 0 ||
      @end_time.wday === 6
  end
end
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- A workbook repository of example test smells and what to do about them
- Smells in software test code: A survey of knowledge in industry and academia

1.1.5.73 Time Sensitive Test

Definition:

- Tests that only pass depending on the time of the day they are executed

Code Example:

References:

Quality attributes

- - Code Example

- - Cause and Effect
- - Frequency
- - Refactoring

- A testing anti-pattern safari

1.1.5.74 Treating Test Code As A Second Class Citizen

Definition:

- Tests with huge code duplication, hardcoded variables, copy-paste segments and several other inefficiencies that would be considered inexcusable if found on the main code.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-patterns of automated testing
- Software Testing Anti-patterns

1.1.5.75 Unclassified Method Category

Definition:

- A test method that is not organized by a method category.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Rule-based Assessment of Test Quality
- Test Smell Detection Tools: A Systematic Mapping Study

1.1.5.76 Unreachable Default

Definition:

- An alt statement contains an else branch while a default is active. The else branch of an alt statement is taken if no other branch is applicable. If a default is active at the same time, its branches come after all branches of the alt statement. Hence the default altstep can never be executed if an else branch is present.

Code Example:

```
testcase myTestcase() runs on MyComponent {
  var default myDefaultVar := activate(myAltstep(t))
  alt {
    [] p.receive(charstring:("foo1")) {
      p.send("ack")
    }
    [] p.receive(charstring:("bar1")) {
      p.send("nack")
    }
    [else] {
      setverdict(fail)
      log("unexpected behavior")
    }
  }
  deactivate(myDefaultVar)
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- An approach to quality engineering of TTCN-3 test specifications
- Pattern-based Smell Detection in TTCN-3 Test Suites
- Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites

1.1.5.77 Unrestricted Imports

Definition:

- A module imports more from another module than needed.

Code Example:

```
module Foo {
  group groupConstants {
  const charstr ing FOO CONST := " foo " ;
    // some other constants . . .
```

(continues on next page)

(continued from previous page)

```
    }

    group groupTypes {
      // type definitions . . .
    }

    group groupComponents {
      // component definitions . . .
    }
  }

module Baz {
  import from Foo all;

  function f ( in charstring s ) return boolean {
    if (FOO_CONST == s ) {
      return true;
    }

    return false;
  }
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- An approach to quality engineering of TTCN-3 test specifications
- Pattern-based Smell Detection in TTCN-3 Test Suites
- Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites

1.1.5.78 Unsuitable Naming

Definition:

- A keyword, macro component, macro event, or primitive component is not properly named, making the script difficult to understand.

References:

Quality attributes

- - Code Example
- - Cause and Effect

- - Frequency
- - Refactoring

- Bad smells and refactoring methods for GUI test script

1.1.5.79 Unused Imports

Definition:

- An import from another module is never used.

Code Example:

```
 module Foo {
   const charstr ing FOO CONST := " foo " ;
 }

 module Bar {
   const charstr ing BAR CONST := " bar " ;
 }

module Baz {
 import from Foo all;
 import from Bar all;

 function f ( in charstring s ) return boolean {
   if (FOO_CONST == s ) {
     return true;
   }

   return false;
 }
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- An approach to quality engineering of TTCN-3 test specifications
- Pattern-based Smell Detection in TTCN-3 Test Suites
- Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites

1.1.5.80 Unused Inputs

Definition:

- Inputs that are controlled by the test

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Automatic generation of smell-free unit tests
- Test Smell Detection Tools: A Systematic Mapping Study

1.1.5.81 Unused Parameter

Definition:

- A parameter is never used within the declaring unit. For in-parameters, the parameter is never read, for out-parameters never defined, for inout-parameters never accessed at all.

Code Example:

```
function f ( in integer i , in integer j ) return integer {
  var integer k := 1 ;
  return i + k ;
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- An approach to quality engineering of TTCN-3 test specifications
- Pattern-based Smell Detection in TTCN-3 Test Suites
- Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites

1.1.5.82 Unused Shared-Fixture Variables

Definition:

- Occurs when a piece of the fixture is never used.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Rule-based Assessment of Test Quality
- Test Smell Detection Tools: A Systematic Mapping Study

1.1.5.83 Unused Variable Definition

Definition:

- A defined variable or in parameter is not read before it becomes undefined.

Also Known As:

- DU Data Flow Anomaly

Code Example:

```
function f ( in integer i , out integer j ) {
  var integer k := 1;
  j := 1;
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- An approach to quality engineering of TTCN-3 test specifications
- Pattern-based Smell Detection in TTCN-3 Test Suites
- Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites

1.1.5.84 Unworldly Test Data

Definition:

- where the test data is in a different style to real-world data e.g. time processing based on epoch milliseconds near 0, rather than on sensible timestamps that would be used in the real world

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.1.5.85 Using Faker

Definition:

- Faker tests become nondeterministic: run - crashed, run again - passed. Such flaky tests are extremely difficult to debug and annoying on CI.

Code Example:

- No code examples yet…

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Using Faker is anti-pattern

1.1.5.86 Verbless And Noun-Full

Definition:

- when a test is named after the concept, but not after the intended behaviour it verifies – behaviour would normally use verbs

References:

Quality attributes

- - Code Example

- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.1.5.87 Wasted Variable Definition

Definition:

- A variable is defined and defined again before it is read.

Also Known As:

- DD Data Flow Anomaly

Code Example:

```
function f ( in integer i , out integer j ) {
  var integer k := 1 ;
  k := i ;
  j := k ;
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- An approach to quality engineering of TTCN-3 test specifications
- Pattern-based Smell Detection in TTCN-3 Test Suites
- Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites

1.2 Dependencies

1.2.1 Dependencies among tests

1.2.1.1 Chain Gang

Definition:

- A couple of tests that must run in a certain order, i.e. one test changes the global state of the system (global variables, data in the database) and the next test(s) depends on it.

Also Known As:

- Order Dependent Tests

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-Patterns - Digital Tapestry
- Smells in software test code: A survey of knowledge in industry and academia
- Unit testing Anti-patterns catalogue

1.2.1.2 Constrained Test Order

Definition:

- Tests expecting to be run in a specific order or expecting information from other test results

Code Example:

```
[TestFixture]
public class IsolationsAntiPatterns
{
  private LogAnalyzer logan;

  [Test]
  public void CreateAnalyzer_BadFileName_ReturnsFalse()
  {
    logan = new LogAnalyzer();

    logan.Initialize();

    bool valid = logan.IsValid("abc");

    Assert.That(valid, Is.False);
  }

  [Test]
  public void CreateAnalyzer_GoodFileName_ReturnsTrue()
  {
    bool valid = logan.IsValid("abcdefg");

    Assert.That(valid, Is.True);
  }

}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Chapter 8. The pillars of good unit tests

1.2.1.3 Coupled Tests

Definition:

- This occurs when testing one scenario affects another; for example, when all the test data are created at once and not rolled back after each scenario and re-created as needed for the next.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Design Patterns for Database API Testing 1: Web Service Saving 2 – Code

1.2.1.4 Coupling Between Test Methods

Definition:

- Test methods (and all tests in general) must be perfectly isolated from each other. This means that changing one test must not affect any others.

Code Example:

```
public final class MetricsTest {
  private File temp;
  private Folder folder;
  @Before
  public void prepare() {
    this.temp = Files.createTempDirectory("test");
    this.folder = new DiscFolder(this.temp);
    this.folder.save("first.txt", "Hello, world!");
    this.folder.save("second.txt", "Goodbye!");
  }
  @After
  public void clean() {
    FileUtils.deleteDirectory(this.temp);
```

(continues on next page)

(continued from previous page)

```
  }
  @Test
  public void calculatesTotalSize() {
    assertEquals(22, new Metrics(this.folder).size());
  }
  @Test
  public void countsWordsInFiles() {
    assertEquals(4, new Metrics(this.folder).wc());
  }
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- A few thoughts on unit test scaffolding
- Refactoring Test Smells With JUnit 5: Why Should Developers Keep Up-to-Date?

1.2.1.5 Dependent Test

Definition:

- Occurs when the test being executed depends on other tests' success. As a result, dependent tests can fail for the wrong reason

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Characterizing High-Quality Test Methods: A First Empirical Study
- JTDog: a Gradle Plugin for Dynamic Test Smell Detection
- On the influence of Test Smells on Test Coverage
- On the test smells detection: an empirical study on the jnose test accuracy
- Test Smell Detection Tools: A Systematic Mapping Study

1.2.1.6 Identity Dodgems

Definition:

- where each test case shares some sort of global resource – perhaps a database, or a singleton data store, so needs to choose identifiers carefully in order to avoid collisions with other tests. Better in this case to use a central ID generator, to avoid accidental collisions, or each test having to be aware of all other tests' choice of IDs.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.2.1.7 Interacting Test Suites

Definition:

- A special case of Interacting Tests where the tests are in different test suites.

Code Example:

```
Suite1.run()--> Green
Suite2.run()--> Green
Suite(Suite1,Suite2).run()--> Test C in Suite2 fails
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Categorising Test Smells
- xUnit test patterns: Refactoring test code

1.2.1.8 Interacting Tests

Definition:

- Tests depend on each other in some way. Note that Interacting Test Suites and Lonely Test are specific variations of Interacting Tests.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Categorising Test Smells
- Quality defects detection in unit tests
- xUnit test patterns: Refactoring test code

1.2.1.9 Lack Of Cohesion Of Test Cases

Definition:

- Occurs if test cases are grouped together in one test suite but are not cohesive.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- PyNose: A Test Smell Detector For Python

1.2.1.10 Lack Of Cohesion Of Test Methods

Definition:

- Cohesion of a class indicates how strongly related and focused the various responsibilities of a class are [4]. Classes with high cohesion facilitate code comprehension and maintenance. Low cohesive methods are smelly because they aggravate reuse, maintainability and comprehension [6], [12]. The smell Lack of Cohesion of test methods (LCOTM) occurs if test methods are grouped together in one test class, but they are not cohesive.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Automated Detection of Test Fixture Strategies and Smells
- Automatic Test Smell Detection Using Information Retrieval Techniques
- Automatic generation of smell-free unit tests
- Just-In-Time Test Smell Detection and Refactoring: The DARTS Project
- Strategies for avoiding text fixture smells during software evolution
- Test Smell Detection Tools: A Systematic Mapping Study

1.2.1.11 Litter Bugs

Definition:

- Each test has a side effect that persists between test cases, often resulting in tests that depend on one another. This is often called "test pollution"

Also Known As:

- Test Pollution

Code Example:

```
# Subject under test
$all_time_logins = 0
class Game
  def self.instance
    @game ||= new
  end

  def initialize
    @players = []
  end

  def add_player(name)
    @players << name
    $all_time_logins += 1
  end

  def player_count
    @players.size
  end
end

# Test
class LitterBugs < SmellTest
  include UnreliableMinitestPlugin
```

(continues on next page)

(continued from previous page)

```ruby
  def setup
    @game = Game.instance
  end

  def test_login_one_player
    @game.add_player("Joe")

    assert_equal 1, @game.player_count
    assert_equal 1, $all_time_logins
  end

  def test_login_two_players
    @game.add_player("Jane")
    @game.add_player("Stef")

    assert_equal 3, @game.player_count
    assert_equal 3, $all_time_logins
  end
end
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- A workbook repository of example test smells and what to do about them

1.2.1.12 Lonely Test

Definition:

- Lonely Test is a special case of Interacting Tests in which a test can be run as part of a suite but cannot be run by itself because it depends on something in a Shared Fixture that was created by another test (e.g. Chained Tests (page X)) or by suite-level fixture setup logic (such as a Setup Decorator.)

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Categorising Test Smells

- xUnit test patterns: Refactoring test code

1.2.1.13 Order Dependent Tests

Definition:

- The tests have to be executed in a certain order due to dependencies between them.

Also Known As:

- Chained Tests, Chain Gang

Code Example:

```
class AnimalTest < Test::Unit::TestCase

  def test_create_record
    a = Animal.create!(name:"Lion")
    assert_not_nil a
  end

  def test_find_record
    a = Animal.find_by_name("Lion")
    assert_not_nil a
  end

end
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- A testing anti-pattern safari
- Categorising Test Smells
- Smells of Testing (signs your tests are bad)

1.2.1.14 Test Pollution

Definition:

- Test that introduces dependencies such as reading/writing a shared resource

Also Known As:

- Litter Bugs

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Automatic generation of smell-free unit tests
- Test Smell Detection Tools: A Systematic Mapping Study

1.2.1.15 Test Run War

Definition:

- Test failures occur at random when several people are running tests simultaneously.

Code Example:

```
Suite.run() --> Test 3 fails
Suite.run() --> Test 3 crashes
Suite.run() --> All tests pass
Suite.run() --> Test 3 fails
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- A survey on test practitioners' awareness of test smells
- An empirical analysis of the distribution of unit test smells and their impact on software maintenance
- An exploratory study of the relationship between software test smells and fault-proneness
- Are test smells really harmful? An empirical study
- Automatic generation of smell-free unit tests
- Categorising Test Smells
- Enhancing developers' awareness on test suites' quality with test smell summaries
- How are test smells treated in the wild? A tale of two empirical studies
- On the interplay between software testing and evolution and its effect on program comprehension
- Refactoring Test Code
- Refactoring Test Smells With JUnit 5: Why Should Developers Keep Up-to-Date?
- Test Smell Detection Tools: A Systematic Mapping Study

- Understanding practitioners' strategies to handle test smells: a multi-method study
- xUnit test patterns: Refactoring test code

1.2.1.16 The Environmental Vandal

Definition:

- A 'unit' test which for various 'requirements' starts spilling out into its environment, using and setting environment variables / ports. Running two of these tests simultaneously will cause 'unavailable port' exceptions etc.

Also Known As:

- The Local Hero

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Unit testing Anti-patterns catalogue

1.2.1.17 The Inhuman Centipede

Definition:

- where a test can only function in sequence with those around it. While there are some situations where this is an expected/necessary evil, it's a terrible default.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.2.1.18 The Leaky Cauldron

Definition:

- similar to Identity Dodgems this is a test which is either sensitive to some state caused by other tests, or which creates a mess that breaks other tests

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.2.1.19 The Parasite

Definition:

- a test which should be written stand-alone, but depends on the running of a previous test

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.2.1.20 The Peeping Tom

Definition:

- A test that, due to the shared resources, can see the result data of another test, and may cause the test to fail even though the system under test is perfectly valid. This has been seen commonly in Fitnesse, where the use of static member variables to hold collections aren't properly cleaned after test execution, often popping up unexpectedly in other test runs.

Also Known As:

- The Uninvited Guests

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-Patterns - Digital Tapestry
- Categorising Test Smells
- Test-Driven Development: TDD Anti-Patterns

1.2.1.21 Unusual Test Order

Definition:

- Tests calling each other explicitly (unusual for unit tests)

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Automatic generation of smell-free unit tests
- Rule-based Assessment of Test Quality
- Test Smell Detection Tools: A Systematic Mapping Study

1.2.2 External dependencies

1.2.2.1 Context Sensitivity

Definition:

- Changing the state or behaviour of the context in which the production code is embedded causes the failure of the test. This is the case, for example, when the code that is changed does not belong to the system under test.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Categorising Test Smells
- xUnit test patterns: Refactoring test code

1.2.2.2 Counting On Spies

Definition:

- occurs when injecting a proxy ob- ject (spy) in place of a dependency (or collaborator), and spying on how the consuming class calls it.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Smells in software test code: A survey of knowledge in industry and academia

1.2.2.3 Data Sensitivity

Definition:

- Data Sensitivity occurs when a test fails because the data being used to test the SUT has been modified. It most commonly occurs when the contents of the test database is changed.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Categorising Test Smells
- xUnit test patterns: Refactoring test code

1.2.2.4 External Data

Definition:

- A test that needs external data (fixtures, files, helper classes, etc.) in order to run can be hard to understand as the reader must file hop to get a clear picture.

Also Known As:

- Mystery Guest

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Smells of Testing (signs your tests are bad)

1.2.2.5 External Dependencies

Definition:

- There are a number of external dependencies that code may need to rely on to work correctly. For example: a specific time or date, a third-party library jar, a file, a database, a network connection, a web container, an application container, randomness, many other dependencies

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Categorising Test Smells
- JUnit Anti-patterns

1.2.2.6 Factories Depending On Database Records

Definition:

- Adding a hard dependency on specific database records in factory definitions leads to build failures in CI environment.

Code Example:

```
factory :active_schedule do
  start_date Date.current - 1.month
  end_date 1.month.since(Date.current)
  processing_status "processed"
  schedule_duration ScheduleDuration.find_by_name("Custom")
end
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Rails Testing Antipatterns: Fixtures and Factories

1.2.2.7 Hidden Dependency

Definition:

- Closely related to the local hero, a unit test that requires some existing data to have been populated somewhere before the test runs. If that data wasn't populated, the test will fail and leave little indication to the developer what it wanted, or why… forcing them to dig through acres of code to find out where the data it was using was supposed to come from.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-Patterns - Digital Tapestry
- Categorising Test Smells
- Smells in software test code: A survey of knowledge in industry and academia
- Tdd antipatterns: Local hero

- Test-Driven Development: TDD Anti-Patterns
- Unit testing Anti-patterns catalogue

1.2.2.8 Hidden Integration Test

Definition:

- A PUT outcome depends on the state of the environment.

Code Example:

```
[PexMethod]
void FileExists(string fileName) {
if (!File.Exists(fileName))
throw new FileNotFoundException();
...
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Parameterized Test Patterns For Effective Testing with Pex

1.2.2.9 Local Only Testing

Definition:

- The recurring coding pattern of only using the local environment to conduct Ansible testing

Code Example:

- No code examples yet…

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- As Code Testing: Characterizing Test Quality in Open Source Ansible Development
- Practitioner Perceptions of Ansible Test Smells

1.2.2.10 Middle Man

Definition:

- A test component (keyword, macro, function) delegates all its tasks to another test component.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Bad smells and refactoring methods for GUI test script
- On the Maintenance of System User Interactive Tests
- Smells in System User Interactive Tests

1.2.2.11 Mystery Guest

Definition:

- A test case that uses external resources that are not managed by a fixture. A drawback of this approach is that the interface to external resources might change over time necessitating an update of the test case, or that those resources might not be available when the test case is run, endangering the deterministic behavior of the test.

Also Known As:

- External Data

Code Example:

```
public void testGetFlightsByFromAirport_OneOutboundFlight_mg() throws Exception {
    loadAirportsAndFlightsFromFile("test-flights.csv");
    // Exercise System
    List flightsAtOrigin = facade.getFlightsByOriginAirportCode( "YYC");
    // Verify Outcome
    assertEquals( 1, flightsAtOrigin.size());
    FlightDto firstFlight = (FlightDto) flightsAtOrigin.get(0);
    assertEquals( "Calgary", firstFlight.getOriginCity());
 }
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- A preliminary evaluation on the relationship among architectural and test smells
- A survey on test practitioners' awareness of test smells
- An Empirical Study into the Relationship Between Class Features and Test Smells
- An Exploratory Study on the Refactoring of Unit Test Files in Android Applications
- An empirical analysis of the distribution of unit test smells and their impact on software maintenance
- An empirical investigation into the nature of test smells
- An exploratory study of the relationship between software test smells and fault-proneness
- Are test smells really harmful? An empirical study
- Assessing diffusion and perception of test smells in scala projects
- Automatic Identification of High-Impact Bug Report by Product and Test Code Quality
- Automatic generation of smell-free unit tests
- Categorising Test Smells
- Detecting redundant unit tests for AspectJ programs
- Enhancing developers' awareness on test suites' quality with test smell summaries
- Handling Test Smells in Python: Results from a Mixed-Method Study
- How are test smells treated in the wild? A tale of two empirical studies
- Investigating Severity Thresholds for Test Smells
- Investigating Test Smells in JavaScript Test Code
- Let's not
- Machine Learning-Based Test Smell Detection
- Mystery Guest
- Obscure Test
- On the Distribution of "Simple Stupid Bugs" in Unit Test Files: An Exploratory Study
- On the Relation of Test Smells to Software Code Quality
- On the diffusion of test smells and their relationship with test code quality of Java projects
- On the diffusion of test smells in automatically generated test code: an empirical study
- On the distribution of test smells in open source Android applications: an exploratory study
- On the influence of Test Smells on Test Coverage
- On the interplay between software testing and evolution and its effect on program comprehension
- On the test smells detection: an empirical study on the jnose test accuracy
- On the use of test smells for prediction of flaky tests
- Rails Testing Antipatterns
- Refactoring Test Code
- Refactoring Test Smells With JUnit 5: Why Should Developers Keep Up-to-Date?
- Revisiting Test Smells in Automatically Generated Tests: Limitations, Pitfalls, and Opportunities
- Scented since the beginning: On the diffuseness of test smells in automatically generated test code

- SoCRATES: Scala radar for test smells
- Software Unit Test Smells
- Test Smell Detection Tools: A Systematic Mapping Study
- TestAXE: Automatically Refactoring Test Smells Using JUnit 5 Features
- TestQ: Exploring Structural and Maintenance Characteristics of Unit Test Suites
- The Relation of Test-Related Factors to Software Quality: A Case Study on Apache Systems
- The secret life of test smells-an empirical study on test smell evolution and maintenance
- To What Extent Can Code Quality be Improved by Eliminating Test Smells?
- Toward static test flakiness prediction: a feasibility study
- Towards Automated Tools for Detecting Test Smells: An Empirical Investigation into the Nature of Test Smells
- What We Know About Smells in Software Test Code
- What the Smell? An Empirical Investigation on the Distribution and Severity of Test Smells in Open Source Android Applications
- Why do builds fail?—A conceptual replication study
- tsDetect: an open source test smells detection tool

1.2.2.12 Programming Paradigms Blend

Definition:

- When there are fields initialized outside the test class (global scope) and used in the test class

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Handling Test Smells in Python: Results from a Mixed-Method Study
- TEMPY: Test Smell Detector for Python

1.2.2.13 Remote Mystery Guest

Definition:

- The recurring coding pattern of using a remote artifact for test play execution, which needs to be accessed via an HTTP/HTTPS-based URL.

Code Example:

- No code examples yet…

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- As Code Testing: Characterizing Test Quality in Open Source Ansible Development
- Practitioner Perceptions of Ansible Test Smells

1.2.2.14 Require External Resources

Definition:

- Test require external resources but can not guarantee their state and availability

Code Example:

```
class Clever
  def districts
    url = "/districts"
    key = "DEMO_KEY"
    `curl -u #{key}: #{host}#{url}`
  end
end

class External < Test::Unit::TestCase
  def test_get_clever_data
    c = Clever.new
    assert_include c.districts, 'Demo'
  end
end
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- A testing anti-pattern safari

1.2.2.15 Resource Leakage

Definition:

- The test uses finite resources. If the tests do not release them in the end, all of the resources are allocated and tests which need the resources begin to fail with time.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Categorising Test Smells
- Refactoring Test Smells With JUnit 5: Why Should Developers Keep Up-to-Date?
- xUnit test patterns: Refactoring test code

1.2.2.16 Resource Optimism

Definition:

- This smell happens when test methods make optimistic assumptions about the existence or the state of external resources like files and databases.
- This smell occurs when a test method makes an optimistic assumption that the external resource (e.g., File), utilized by the test method, exists.

Code Example:

```
@Test
public void saveImage_noImageFile_ko() throws IOException {
  File outputFile = File.createTempFile("prefix", "png", new File("/tmp"));
  ProductImage image = new ProductImage("01010101010101", ProductImageField.FRONT,␣
↪outputFile);
  Response response = serviceWrite.saveImage(image.getCode(), image.getField(), image.
↪getImguploadFront(), image.getImguploadIngredients(), image.getImguploadNutrition()).
↪execute();
  assertTrue(response.isSuccess());
  assertThatJson(response.body())
      .node("status")
          .isEqualTo("status not ok");
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect

- - Frequency
- - Refactoring

- A preliminary evaluation on the relationship among architectural and test smells
- A survey on test practitioners' awareness of test smells
- An Exploratory Study on the Refactoring of Unit Test Files in Android Applications
- An empirical analysis of the distribution of unit test smells and their impact on software maintenance
- An exploratory study of the relationship between software test smells and fault-proneness
- Are test smells really harmful? An empirical study
- Automatic Identification of High-Impact Bug Report by Product and Test Code Quality
- Automatic generation of smell-free unit tests
- Categorising Test Smells
- Enhancing developers' awareness on test suites' quality with test smell summaries
- Handling Test Smells in Python: Results from a Mixed-Method Study
- How are test smells treated in the wild? A tale of two empirical studies
- Investigating Severity Thresholds for Test Smells
- Investigating Test Smells in JavaScript Test Code
- Machine Learning-Based Test Smell Detection
- On the Distribution of "Simple Stupid Bugs" in Unit Test Files: An Exploratory Study
- On the Relation of Test Smells to Software Code Quality
- On the diffusion of test smells and their relationship with test code quality of Java projects
- On the diffusion of test smells in automatically generated test code: an empirical study
- On the distribution of test smells in open source Android applications: an exploratory study
- On the influence of Test Smells on Test Coverage
- On the interplay between software testing and evolution and its effect on program comprehension
- On the test smells detection: an empirical study on the jnose test accuracy
- On the use of test smells for prediction of flaky tests
- Refactoring Test Code
- Refactoring Test Smells With JUnit 5: Why Should Developers Keep Up-to-Date?
- Refactoring Test Smells: A Perspective from Open-Source Developers
- Revisiting Test Smells in Automatically Generated Tests: Limitations, Pitfalls, and Opportunities
- Scented since the beginning: On the diffuseness of test smells in automatically generated test code
- Software Unit Test Smells
- Test Smell Detection Tools: A Systematic Mapping Study
- The Relation of Test-Related Factors to Software Quality: A Case Study on Apache Systems
- The secret life of test smells-an empirical study on test smell evolution and maintenance

- To What Extent Can Code Quality be Improved by Eliminating Test Smells?
- Toward static test flakiness prediction: a feasibility study
- Understanding practitioners' strategies to handle test smells: a multi-method study
- What the Smell? An Empirical Investigation on the Distribution and Severity of Test Smells in Open Source Android Applications
- tsDetect: an open source test smells detection tool
- xUnit test patterns: Refactoring test code

1.2.2.17 Stinky Synchronization Syndrome

Definition:

- The failure to use proper synchronization/wait points in the GUI-based testing of web applications using tools such as Selenium.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Smells in software test code: A survey of knowledge in industry and academia

1.2.2.18 Stinky Synchronization

Definition:

- The test fails to use proper synchronization points with the system under test

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- On the Maintenance of System User Interactive Tests
- Smells in System User Interactive Tests
- Smells in software test code: A survey of knowledge in industry and academia

1.2.2.19 Tests Depend On Something Outside Of The Test Suite

Definition:

- occur when there is something the test needs to be able to run but it is not mocked either because it wasn't missed or is not possible to mock

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-Patterns In Unit Testing

1.2.2.20 The Local Hero

Definition:

- A test case that is dependent on something specific to the development environment it was written on, in order to run. The result is that the test passes on development boxes, but fails when someone attempts to run it elsewhere.

Also Known As:

- Wait and See, The Environmental Vandal

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-Patterns - Digital Tapestry
- Categorising Test Smells
- Smells in software test code: A survey of knowledge in industry and academia
- Tdd antipatterns: Local hero
- Test-Driven Development: TDD Anti-Patterns
- Unit Testing Anti-Patterns, Full List
- Unit testing Anti-patterns catalogue

1.2.2.21 The Operating System Evangelist

Definition:

- A unit test that relies on a specific operating system environment to be in place in order to work. A good example would be a test case that uses the newline sequence for Windows in an assertion, only to break when run on Linux.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-Patterns - Digital Tapestry
- Categorising Test Smells
- TDD and anti-patterns - Chapter 5
- Test-Driven Development: TDD Anti-Patterns

1.2.2.22 Web-Browsing Test

Definition:

- To be run the test needs to establish a connection to the internet.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Categorising Test Smells

1.3 Design related

1.3.1 Not using test patterns

1.3.1.1 Autogeneration

Definition:

- Auto-generated tests that test methods instead of behavior

Code Example:

```java
public void testSetGetTimestamp() throws Exception {
  // JUnitDoclet begin method setTimestamp getTimestamp
  java.util.Calendar[] tests = {new GregorianCalendar(), null};

  for (int i = 0; i < tests.length; i++) {
    adapter.setTimestamp(tests[i]);
    assertEquals(tests[i], adapter.getTimestamp());
  }
  // JUnitDoclet end method setTimestamp getTimestamp
}
public void testSetGetParam() throws Exception {
  // JUnitDoclet begin method setParam getParam
  String[] tests = {"a", "aaa", "---", "23121313", "", null};

  for (int i = 0; i < tests.length; i++) {
    adapter.setParam(tests[i]);
    assertEquals(tests[i], adapter.getParam());
  }
  // JUnitDoclet end method setParam getParam
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Bad tests, good tests

1.3.1.2 Constructor Initialization

Definition:

- Ideally, the test suite should not have a constructor. Initialization of fields should be in the setUp() method. Developers who are unaware of the purpose of setUp() method would give rise to this smell by defining a constructor for the test suite.

Code Example:

```
public class TagEncodingTest extends BrambleTestCase {
        private final CryptoComponent crypto;
        private final SecretKey tagKey;
        private final long streamNumber = 1234567890;

        public TagEncodingTest() {
                crypto = new CryptoComponentImpl(new TestSecureRandomProvider());
                tagKey = TestUtils.getSecretKey();
        }

        @Test
        public void testKeyAffectsTag() throws Exception {
                Set set = new HashSet<>();
                for (int i = 0; i < 100; i++) {
                        byte[] tag = new byte[TAG_LENGTH];
                        SecretKey tagKey = TestUtils.getSecretKey();
                        crypto.encodeTag(tag, tagKey, PROTOCOL_VERSION, streamNumber);
                        assertTrue(set.add(new Bytes(tag)));
                }
        }
...
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- A survey on test practitioners’ awareness of test smells
- An Exploratory Study on the Refactoring of Unit Test Files in Android Applications
- Automatic Identification of High-Impact Bug Report by Product and Test Code Quality
- Automatic generation of smell-free unit tests
- Developers perception on the severity of test smells: an empirical study
- Handling Test Smells in Python: Results from a Mixed-Method Study
- How are test smells treated in the wild? A tale of two empirical studies
- On the Distribution of “Simple Stupid Bugs” in Unit Test Files: An Exploratory Study

- On the diffusion of test smells and their relationship with test code quality of Java projects
- On the distribution of test smells in open source Android applications: an exploratory study
- On the influence of Test Smells on Test Coverage
- On the test smells detection: an empirical study on the jnose test accuracy
- On the use of test smells for prediction of flaky tests
- PyNose: A Test Smell Detector For Python
- Software Unit Test Smells
- Test Smell Detection Tools: A Systematic Mapping Study
- The secret life of test smells-an empirical study on test smell evolution and maintenance
- Understanding Testability and Test Smells
- What the Smell? An Empirical Investigation on the Distribution and Severity of Test Smells in Open Source Android Applications
- tsDetect: an open source test smells detection tool

1.3.1.3 Contortionist Testing

Definition:

- this is really a design smell. You're probably adding tests after the code was written and are required to bend over backwards to construct those tests owing to poorly designed code. This especially involves NEEDing to use mocking of static functions or types.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.3.1.4 Disorder

Definition:

- The sequence of elements within a module does not conform to a given order. A preferred ordering could be:
  1. imports
  2. module parameters
  3. data types
  4. port types
  5. component types
  6. templates

7. functions
8. altsteps
9. test cases
10. control part

Code Example:

```
function f() {
  //...
}

type record exampleRecordType {
  //...
}

template exampleRecordType t :=  {
  //...
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- An approach to quality engineering of TTCN-3 test specifications
- Pattern-based Smell Detection in TTCN-3 Test Suites
- Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites

1.3.1.5 I'Ll Believe It When I See Some Flashing Guis

Definition:

- An unhealthy fixation/obsession with testing the app via its GUI 'just like a real user'. Testing business rules through the GUI is a terrible form of coupling. If you write thousands of tests through the GUI, and then change your GUI, thousands of tests break. Rather, test only GUI things through the GUI, and couple the GUI to a dummy system instead of the real system, when you run those tests. Test business rules through an API that doesn't involve the GUI.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency

- - Refactoring

- Anti-Patterns - Digital Tapestry
- Unit testing Anti-patterns catalogue

1.3.1.6 Missing Log

Definition:

- setverdict is used with verdict inconc or fail, but without calling log. Inconclusive or unsuccessful test verdicts should be logged, because this helps discovering the reasons for the failure. However, this smell should be classified weak compared to other smells.

Code Example:

```
testcase exampleTestCase () runs on ExampleComponent {
  timer t_guard;
  //...
  t_guard.start(10.0);
  alt {
    [] pt.receive(a_MessageOne){
      t_guard.stop
      setverdict(pass)
      pt.send(a_MessageTwo);
    }
    [] any port.receive {
      repeat;
    }
    [] t_guard.timeout {
      setverdict(fail)
      stop;
    }
  }
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- An approach to quality engineering of TTCN-3 test specifications
- Pattern-based Smell Detection in TTCN-3 Test Suites
- Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites

1.3.1.7 Narcissistic

Definition:

- The test uses the first person pronoun "I" to refer to its actors and does not uniquely qualify those actors

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- On the Maintenance of System User Interactive Tests
- Smells in System User Interactive Tests

1.3.1.8 No Clear Structure Within The Test

Definition:

- Having no consistent test structure, harming readability

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-patterns of automated testing

1.3.1.9 No Structure When Creating Test Cases

Definition:

- Test code is also code and should have a structure to it so that it can be easily read and altered if needed. Not having a structure to test code makes it hard to understand and maintain. A simple structure is to organize the code into the different stages of the testing process.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency

- - Refactoring

- Anti-Patterns In Unit Testing

1.3.1.10 Not Using Page-Objects

Definition:

- Page objects are just a design pattern to ensure automated UI tests use reusable, modular code. Not using them, eg, writing WebDriver code directly in step definitions, means any changes to your UI will require updates in lots of different places instead of the one 'page' class.

Code Example:

```
[When(@'I buy some '(.*)' tea')]
public void WhenIBuySomeTea(string typeOfTea)
{
  Driver.FindElement(By.Id('tea-'+typeOfTea)).Click();
  Driver.FindElement(By.Id('buy')).Click();
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Five automated acceptance test anti-patterns

1.3.1.11 So The Automation Tool Wrote This Crap

Definition:

- where the code of the test was ejected out of the guts of a code-generator built from a test GUI. This may not be a real issue, if the code isn't then modified and updated. However, if this code is actively developed, then de-robotise it.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.3.1.12 Test::Class Hierarchy

Definition:

- When test classes are not independent and test methods are inherited

Code Example:

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test::Class Hierarchy Is an Antipattern

1.3.1.13 Testing Business Rules Through Ui

Definition:

- business rules should be tested regardless of the UI being used. Of course, end-to-end testing is an exception

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-patterns of automated testing

1.3.1.14 The Turing Test

Definition:

- A testcase automagically generated by some expensive tool that has many, many asserts gleaned from the class under test using some too-clever-by-half data flow analysis. Lulls developers into a false sense of confidence that their code is well tested, absolving them from the responsibility of designing and maintaining high quality tests. If the machine can write the tests for you, why can't it pull its finger out and write the app itself!

References:

Quality attributes

- - Code Example
- - Cause and Effect

- - Frequency
- - Refactoring

- Unit testing Anti-patterns catalogue

1.3.1.15 Unclassified Method Category

Definition:

- A test method that is not organized by a method category.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Rule-based Assessment of Test Quality
- Test Smell Detection Tools: A Systematic Mapping Study

1.3.1.16 Undefined Test

Definition:

- It occurs when the name of a test method is not prefixed with “test”.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Handling Test Smells in Python: Results from a Mixed-Method Study
- TEMPY: Test Smell Detector for Python

1.3.1.17 Unknown Test

Definition:

- An assertion statement is used to declare an expected boolean condition for a test method. By examining the assertion statement it is possible to understand the purpose of the test method. However, It is possible for a test method to written sans an assertion statement, in such an instance JUnit will show the test method as passing if the statements within the test method did not result in an exception, when executed. New developers to the project will find it difficult in understanding the purpose of such test methods (more so if the name of the test method is not descriptive enough).

Code Example:

```
@Test
public void hitGetPOICategoriesApi() throws Exception {
    POICategories poiCategories = apiClient.getPOICategories(16);
    for (POICategory category : poiCategories) {
      System.out.println(category.name() + ": " + category);
    }
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- An Exploratory Study on the Refactoring of Unit Test Files in Android Applications
- Automatic Identification of High-Impact Bug Report by Product and Test Code Quality
- Automatic generation of smell-free unit tests
- Characterizing High-Quality Test Methods: A First Empirical Study
- Developers perception on the severity of test smells: an empirical study
- Handling Test Smells in Python: Results from a Mixed-Method Study
- Investigating Test Smells in JavaScript Test Code
- On the Distribution of "Simple Stupid Bugs" in Unit Test Files: An Exploratory Study
- On the diffusion of test smells and their relationship with test code quality of Java projects
- On the distribution of test smells in open source Android applications: an exploratory study
- On the influence of Test Smells on Test Coverage
- On the test smells detection: an empirical study on the jnose test accuracy
- On the use of test smells for prediction of flaky tests
- PyNose: A Test Smell Detector For Python
- Pytest-Smell: a smell detection tool for Python unit tests
- Software Unit Test Smells

- TEMPY: Test Smell Detector for Python
- Test Smell Detection Tools: A Systematic Mapping Study
- The secret life of test smells-an empirical study on test smell evolution and maintenance
- Understanding practitioners' strategies to handle test smells: a multi-method study
- What the Smell? An Empirical Investigation on the Distribution and Severity of Test Smells in Open Source Android Applications
- tsDetect: an open source test smells detection tool

1.4 Issues in test steps

1.4.1 Issues in assertions

1.4.1.1 7 Layer Testing

Definition:

- Numerous tests depend on the functionality of a single unit, typically incidentally. A single change in the code often breaks many tests in the build. Often exhibits itself when a team finds that even trivial changes to a system results in exorbitant effort to get back to a green build.

Code Example:

```
// Subject under test
var _ = require('lodash')
function annualProfit (year) {
  return _.sumBy(_.range(1, 13), function (month) {
    return monthlyProfit(year, month)
  })
}

function monthlyProfit (year, month) {
  return _.sumBy(_.range(1, 32), function (day) {
    return dailyProfit(year, month, day)
  })
}

function dailyProfit (year, month, day) {
  var transactions = repo.find({year: year, month: month, day: day})
  return _.sumBy(transactions, function (transaction) {
    return transactionProfit(transaction)
  })
}

function transactionProfit (transaction) {
  return Math.round(transaction.price - transaction.cost)
}

// Test
module.exports = {
  computesAnnualProfit: function () {
```

(continues on next page)

(continued from previous page)

```
    repo.saveTransactions([
      {year: 2016, month: 3, day: 14, price: 55.44, cost: 80.30},
      {year: 2016, month: 6, day: 20, price: 9.33, cost: 4.00},
      {year: 2016, month: 12, day: 31, price: 112.48, cost: 100.24},
      // Bad year:
      {year: 2015, month: 3, day: 14, price: 999, cost: 0}
    ])

    var result = annualProfit(2016)

    assert.equal(result, -8)
  },
  computesMonthlyProfit: function () {
    repo.saveTransactions([
      {year: 2016, month: 5, day: 1, price: 108.99, cost: 70.45},
      {year: 2016, month: 5, day: 15, price: 208.13, cost: 133.55},
      {year: 2016, month: 5, day: 31, price: 90.00, cost: 80.03},
      // Bad month:
      {year: 2016, month: 6, day: 14, price: 999, cost: 0}
    ])

    var result = monthlyProfit(2016, 5)

    assert.equal(result, 124)
  },
  computesDailyProfit: function () {
    repo.saveTransactions([
      {year: 2016, month: 5, day: 12, price: 19.44, cost: 18.11},
      {year: 2016, month: 5, day: 12, price: 21.40, cost: 22.01},
      {year: 2016, month: 5, day: 12, price: 998.10, cost: 907.20},
      // Bad day:
      {year: 2016, month: 5, day: 1, price: 999, cost: 0}
    ])

    var result = dailyProfit(2016, 5, 12)

    assert.equal(result, 91)
  },
  computesTransactionProfit: function () {
    var transaction = {price: 33.22, cost: 20.11}

    var result = transactionProfit(transaction)

    assert.equal(result, 13)
  },
  afterEach: function () {
    repo.reset()
  }
}

// Fake production implementations to simplify example test of subject
var repo = {
```

(continues on next page)

(continued from previous page)

```
  __transactions: [],
  reset: function () {
    repo.__transactions = []
  },
  saveTransactions: function (transactions) {
    repo.__transactions.push.apply(repo.__transactions, transactions)
  },
  find: function (criteria) {
    return _.filter(repo.__transactions, criteria)
  }
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- A workbook repository of example test smells and what to do about them

1.4.1.2 Asserting Pre-Condition And Invariants

Definition:

- Using the same API to express pre-conditions (i.e. argument validation), post-conditions, invariants, and assertions.

Code Example:

```
public void Parse(string input) {
  // pre-condition
  Debug.Assert(input != null, "invalid argument");
  ...
  // invariant
  Debug.Assert(condition, "this should not happen");
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Parameterized Test Patterns For Effective Testing with Pex

1.4.1.3 Assertion Diversion

Definition:

- Where the wrong sort of assert is used, thus making a test failure harder to understand

Code Example:

```
Boolean isValid = false;
if (actualResult.contains("foo")) {
    isValid = true;
}
assertEquals(true, isValid)
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.4.1.4 Assertion Mismatch Scenario

Definition:

- Test cases that do not correctly assert the expected behavior given program specifications.

Code Example:

- No code examples yet…

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Improving Student Testing Practices through a Lightweight Checklist Intervention.

1.4.1.5 Assertion-Free

Definition:

- A test case without assertion

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- An Empirical Study into the Relationship Between Class Features and Test Smells

1.4.1.6 Assertionless Test

Definition:

- A test that does not contain at least one valid assertion is not a real test as it does only execute plain source-code, but never assert any data, state or functionality.
- Pretending to assert data and functionality, but does not

Also Known As:

- Lying Test, The Line Hitter, No Assertions

Code Example:

```
// Smalltalk
ICCreateCalendar>>TesttestCreatingSeveralCalendars
  self addCalendarWithName: 'new Calendar 1'.
  self addCalendarWithName: 'new Calendar 2'.
  self addCalendarWithName: 'new Calendar 3'.
  self addCalendarWithName: 'new Calendar 1'.
  self addCalendarWithName: 'new Calendar 2'.
  self addCalendarWithName: 'new Calendar 3'.
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Assessing test quality - TestLint
- Rule-based Assessment of Test Quality

- Test Smell Detection Tools: A Systematic Mapping Study
- Test Smell Detection Tools: A Systematic Mapping Study
- TestQ: Exploring Structural and Maintenance Characteristics of Unit Test Suites
- What We Know About Smells in Software Test Code

1.4.1.7 Assertions Should Be Merciless

Definition:

- Tests whose assertions do not prove the test method works correctly

Code Example:

```
@Test
public void shouldRemoveEmailsByState() {
  //given
  Email pending = createAndSaveEmail("pending","content pending",
  "abc@def.com", Email.PENDING);
  Email failed = createAndSaveEmail("failed","content failed",
  "abc@def.com", Email.FAILED);
  Email sent = createAndSaveEmail("sent","content sent",
  "abc@def.com", Email.SENT);
  //when
  emailDAO.removeByState(Email.FAILED);
  //then
  assertThat(emailDAO.findAll()).doesNotContain(failed);
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Bad tests, good tests

1.4.1.8 Blinkered Assertions

Definition:

- where the assertions are blind to the fact that the whole answer is wrong, because they're focusing on a subset of the detail.

References:

Quality attributes

- - Code Example

- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.4.1.9 Brittle Assertion

Definition:

- A test method that has assertions that check data input

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Automatic generation of smell-free unit tests
- Test Smell Detection Tools: A Systematic Mapping Study

1.4.1.10 Brittle Test

Definition:

- UI tests containing procedural test code, duplicated steps and magic values

Also Known As:

- Fragile Test

Code Example:

```
class BrittleTest
{
    [Test]
    public void Can_buy_an_Album_when_registered()
    {
        var driver = Host.Instance.Application.Browser;
        driver.Navigate().GoToUrl(driver.Url);
        driver.FindElement(By.LinkText("Admin")).Click();
        driver.FindElement(By.LinkText("Register")).Click();
        driver.FindElement(By.Id("UserName")).Clear();
        driver.FindElement(By.Id("UserName")).SendKeys("HJSimpson");
        driver.FindElement(By.Id("Password")).Clear();
        driver.FindElement(By.Id("Password")).SendKeys("!2345Qwert");
        driver.FindElement(By.Id("ConfirmPassword")).Clear();
        driver.FindElement(By.Id("ConfirmPassword")).SendKeys("!2345Qwert");
```

(continues on next page)

(continued from previous page)

```
        driver.FindElement(By.CssSelector("input[type=\"submit\"]")).Click();
        driver.FindElement(By.LinkText("Disco")).Click();
        driver.FindElement(By.CssSelector("img[alt=\"Le Freak\"]")).Click();
        driver.FindElement(By.LinkText("Add to cart")).Click();
        driver.FindElement(By.LinkText("Checkout >>")).Click();
        driver.FindElement(By.Id("FirstName")).Clear();
        driver.FindElement(By.Id("FirstName")).SendKeys("Homer");
        driver.FindElement(By.Id("LastName")).Clear();
        driver.FindElement(By.Id("LastName")).SendKeys("Simpson");
        driver.FindElement(By.Id("Address")).Clear();
        driver.FindElement(By.Id("Address")).SendKeys("742 Evergreen Terrace");
        driver.FindElement(By.Id("City")).Clear();
        driver.FindElement(By.Id("City")).SendKeys("Springfield");
        driver.FindElement(By.Id("State")).Clear();
        driver.FindElement(By.Id("State")).SendKeys("Kentucky");
        driver.FindElement(By.Id("PostalCode")).Clear();
        driver.FindElement(By.Id("PostalCode")).SendKeys("123456");
        driver.FindElement(By.Id("Country")).Clear();
        driver.FindElement(By.Id("Country")).SendKeys("United States");
        driver.FindElement(By.Id("Phone")).Clear();
        driver.FindElement(By.Id("Phone")).SendKeys("2341231241");
        driver.FindElement(By.Id("Email")).Clear();
        driver.FindElement(By.Id("Email")).SendKeys("chunkylover53@aol.com");
        driver.FindElement(By.Id("PromoCode")).Clear();
        driver.FindElement(By.Id("PromoCode")).SendKeys("FREE");
        driver.FindElement(By.CssSelector("input[type=\"submit\"]")).Click();
        Assert.IsTrue(driver.PageSource.Contains("Checkout Complete"));
    }
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Maintainable automated ui tests
- Smells of Testing (signs your tests are bad)

1.4.1.11 Brittle Ui Tests

Definition:

- Tests having fixed delays, bad selectors and targeting elements, and difficult investigating failures

Code Example:

```
driver.Url = "http://somedomain/url_that_delays_loading";
Thread.Sleep(5000);
IWebElement myDynamicElement = driver.FindElement(By.Id("someDynamicElement"));
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Tips to avoid brittle ui tests

1.4.1.12 Broad Assertion

Definition:

- Assertions that do not compare exact content and are, therefore, broad

Code Example:

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Developer test anti-patterns by lasse koskela

1.4.1.13 Bumbling Assertions

Definition:

- Where there was a more articulate assertion available, but we chose a less sophisticated one and kind of got the message across. E.g. testing exceptions the hard way, or using equality check on list size, rather than a list size assertion.

Code Example:

```
Optional<Foo> result = service.getFoos(123);
assertNotNull(result);
assertThat(result).isNotEmpty();
assertThat(result.getBar()).isNotNull();
assertThat(result.getBar()).hasSize(1);
assertThat(result.getBar().get(0)).isEqualTo("buzz");
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.4.1.14 Calculating Expected Results On The Fly

Definition:

- Any automated test that performs a comparison needs to know the expected results. If you believe that automatically calculating expected results is for you then I would at least consider separating the code that calculates the expected results from the code that performs the actual test.

Code Example:

```
public void CheckWeightCategory_Version1()
{
    // For each item in the file, check it has been labelled correctly
    foreach (Item currentItem in testFile)
    {
        // Navigate to the item tracking page on the website
        selenium.Open("/ItemTrackingPage.html");

      // Get the weight category for the current item
        string acutalWeightCategory = selenium.GetText("weightCategory" + currentItem.
↪ID);

        // Calculate the expected result
        string expectedWeightCategory;

```

(continues on next page)

(continued from previous page)

```
        // If an item weighs more than 50kg then mark the item as a two-man lift,
        // otherwise mark the item as a one-man lift
        if (currentItem.weight > 50)
        {
            expectedWeightCategory = "two-man lift";
        }
        else
        {
            expectedWeightCategory = "one-man lift";
        }

        // Compare actual to expected result
        Assert.AreEqual(acutalWeightCategory, expectedWeightCategory);
    }
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- How test automation with selenium can fail

1.4.1.15 Celery Data

Definition:

- where the data read from the system under test is in a format which is hard to make meaningful assertions on – for example raw JSON Strings.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.4.1.16 Circumstantial Evidence

Definition:

- where the assertions are looking at things which are not direct proof of the behaviour

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.4.1.17 Commented Code In The Test

Definition:

- The test passes but some assertions or parts of the setup, which would cause a test failure, are commented out

Also Known As:

- Under-the-carpet Failing Assertion

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Categorising Test Smells
- Smells of Testing (signs your tests are bad)

1.4.1.18 Commented Test

Definition:

- Test contents are commented so that it passes

Code Example:

```
class SystemAdminSmokeTest extends GroovyTestCase {
  void testSmoke() {
    // do not remove below code
    // def ds = new org.h2.jdbcx.JdbcDataSource(
```

(continues on next page)

(continued from previous page)

```
    // URL: 'jdbc:h2:mem:test;DB_CLOSE_DELAY=-1;MODE=Oracle',
    // user: 'sa', password: '')
    //
    // def jpaProperties = new Properties()
    // jpaProperties.setProperty(
    // 'hibernate.cache.use_second_level_cache', 'false')
    // jpaProperties.setProperty(
    // 'hibernate.cache.use_query_cache', 'false')
    //
    // def emf = new LocalContainerEntityManagerFactoryBean(
    // dataSource: ds, persistenceUnitName: 'my-domain',
    // jpaVendorAdapter: new HibernateJpaVendorAdapter(
    // database: Database.H2, showSql: true,
    // generateDdl: true), jpaProperties: jpaProperties)
    ...
  }
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Bad tests, good tests

1.4.1.19 Complex Assertions

Definition:

- The assertions in a test require many lines of code to implement. At first blush, since test-scoped logic is typically itself untested, this can be risky. Additionally, multi-line assertions are typically harder to read—both in terms of what they're doing and what they intend to say about the subject's behavior.

Code Example:

```
// Subject under test
var _ = require('lodash')
function incrementAge (people) {
  return _(_.cloneDeep(people)).map(function (person) {
    if (_.isNumber(person.age)) {
      person.age += 1
    }
    if (_.isArray(person.kids)) {
      person.kids = incrementAge(person.kids)
    }
    return person
```

(continues on next page)

(continued from previous page)

```
  }).shuffle().value()
}

// Test
module.exports = {
  incrementsSinglePersonAge: function () {
    var people = [
      {name: 'Jane', age: 39},
      {name: 'John', age: 99}
    ]
    var results = incrementAge(people)

    var jane = _.find(results, function (person) { return person.name === 'Jane' })
    assert.equal(jane.age, 40)
    var john = _.find(results, function (person) { return person.name === 'John' })
    assert.equal(john.age, 100)
  },
  incrementsKidsAgeToo: function () {
    var people = [
      {
        name: 'Joe',
        age: 42,
        kids: [
          {name: 'Jack', age: 8},
          {name: 'Jill', age: 7}
        ]}
    ]

    var results = incrementAge(people)

    var jack = _.find(results[0].kids, function (person) {
      return person.name === 'Jack'
    })
    assert.equal(jack.age, 9)
    var jill = _.find(results[0].kids, function (person) {
      return person.name === 'Jill'
    })
    assert.equal(jill.age, 8)
  }
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- A workbook repository of example test smells and what to do about them

1.4.1.20 Conspiracy Of Silence

Definition:

- When assertions in tests are failing with almost no clue why

Code Example:

```
Expected: true
Actual: false
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-patterns in test automation

1.4.1.21 Early Returning Test

Definition:

- A test method that returns a value too early which may drop assertions

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Rule-based Assessment of Test Quality
- Test Smell Detection Tools: A Systematic Mapping Study

1.4.1.22 Equality Sledgehammer Assertion

Definition:

- where the interesting behaviour we are trying to prove is a subset of asserting the equality of everything, for example just knowing the count would be enough, but we assert all values in all rows – don't go too far, through, and end up with a Blinkered Assertion – the likely cause of this smell is lack of imagination when writing an assertion and just landing on equals

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.4.1.23 Fantasy Tests

Definition:

- Passing tests of code that wouldn't actually work in production, usually as a result of a stub returning a response that's substantially different from how a real instance would behave.

Code Example:

```ruby
# Subject under test
class Authorizer
  def initialize(authenticator)
    @authenticator = authenticator
  end

  def roles(user, password)
    if @authenticator.login(user: user, password: password)
      ["admin", "developer", "manager"]
    else
      []
    end
  end
end

# Test
class Fantasy < SmellTest
  def setup
    @authenticator = Minitest::Mock.new(Authenticator.new)
    @subject = Authorizer.new(@authenticator)
    super
  end

  def test_all_roles_if_authenticated
    @authenticator.expect(:login, true, [{user: "hi", password: "bye"}])

    result = @subject.roles("hi", "bye")

    assert_equal ["admin", "developer", "manager"], result
  end

  def test_no_roles_if_not_authenticated
```

(continues on next page)

(continued from previous page)

```
    @authenticator.expect(:login, false, [{user: "hi", password: "no!"}])

    result = @subject.roles("hi", "no!")

    assert_equal [], result
  end
end

# Fake production implementations to simplify example test of subject
class Authenticator
  def login(credentials)
    if !credentials[:password] || !credentials[:"2fa"]
      raise "Both password and two-factor auth token are now required!"
    end
    return true
  end
end
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- A workbook repository of example test smells and what to do about them
- Smells in software test code: A survey of knowledge in industry and academia

1.4.1.24 Fragile Test

Definition:

- Tests which seem to break when they shouldn't

Also Known As:

- Brittle Test

Code Example:

```
describe Post do
  let(:user) { build_stubbed(:user) }
  let(:title) { "Example Post" }
  subject(:post) { build_stubbed(:post, user: user, title: title) }

  # ...

  context "with an author" do
    let(:user) { build_stubbed(:user, name: "Willy") }
```

(continues on next page)

(continued from previous page)

```
    it "returns the author's name" do
      expect(post.author_name).to eq("Willy")
    end
  end
end
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Inspecting Automated Test Code: A Preliminary Study

1.4.1.25 Fuzzy Assertions

Definition:

- Where lack of control for the system under test, causes us not to be able to predict the exact outcome, leading to fuzzy or partial matching in our assertions

Code Example:

```
readUser = system.retrieveUser(id)

//fuzzy match
assert(user).matches([
    { obj.firstName == 'John' },
    { obj.lastName == 'Smith' },
    { obj.id instanceof UUID },
    { obj.creationDate - now() < inSeconds(5) }
])
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.4.1.26 Happenstance Testing

Definition:

- where assertions are trying to lock down implementation details that are not directly important and might validly change, for example the exact wording of an exception

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.4.1.27 Inadequate Assertion

Definition:

- Every update to the execution state must eventually be verified in the assertions. In principle, assertions should verify the correctness of all updates to the object/program state, otherwise the strength of the test oracles is considered not enough to guard the program against faults.

Code Example:

```
public class ClassUnderTest{
    private int data1;
    private int data2;
    public int getData1() { return data1;}
    public int getData2() { return data2;}

    public void method1(int flag){
        if(flag>0){
            this.data2 =  2; //define data2
        }
        this.data1 = 4;      //define data1


public class ClassUnderTest extends TestCase{
    public void testMethod1(){
        ClassUnderTest cut = new ClassUnderTest;
        int testInput = 1;
        cut.method1(testInput);

        assertTrue(cut.getData1()==4); // Is this assertion adequate enough?
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- On adequacy of assertions in automated test suites: an empirical investigation

1.4.1.28 Inappropriate Assertions

Definition:

- An inappropriate assertion is being used. For example, assertTrue is used to check the equality of values instead of assertEquals.

Also Known As:

- Wrong Assert

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Categorising Test Smells

1.4.1.29 Incidental Coverage

Definition:

- Is it enough to know that your test suite encounters every line of code? Or don't you want to be sure that it exercises every line? If you simply encounter the line without asserting that it produces the correct results, are you any better off?

Code Example:

```ruby
require File.dirname(__FILE__) + '/../test_helper'

class ProductsControllerTest < ActionController::TestCase
  def test_should_get_index
    get :index
  end

  # ... remaining tests omitted
end
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Testing anti-patterns: How to fail with 100% test coverage

1.4.1.30 Invisible Assertions

Definition:

- A test which lacks any explicit assertions, leading future readers in the potentially frustrating position of puzzling over the intention of the test.

Code Example:

```
require "helper"

# Subject under test
def is_21?(person)
  if person.age < 21
    raise "Sorry, adults only!"
  end
end

# Test
class InvisibleAssertions < SmellTest
  def test_is_21
    person = OpenStruct.new(age: 21)

    is_21?(person)
  end

  def test_is_under_age
    person = OpenStruct.new(age: 20)

    assert_raises { is_21?(person) }
  end
end
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- A workbook repository of example test smells and what to do about them

1.4.1.31 Likely Ineffective Object-Comparison

Definition:

- A test that performs a comparison between objects will never fail.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Automatic generation of smell-free unit tests
- Rule-based Assessment of Test Quality
- Test Smell Detection Tools: A Systematic Mapping Study

1.4.1.32 Line Hitter

Definition:

- At first glance, the tests cover everything and code coverage tools confirm it with 100%, but in reality the tests merely hit the code, without doing any output analysis.

Also Known As:

- Assertionless Test

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Unit Testing Anti-Patterns, Full List
- Unit testing Anti-patterns catalogue

1.4.1.33 Martini Assertion

Definition:

- where we look for the desired outcome anytime, any place, anywhere – often caused by Celery Data and in some ways the opposite of the Equality Sledgehammer. Rather than look at a specific field to see if it's right, we look for some data in the WHOLE output.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.4.1.34 Missed Fail Rotten Green Test

Definition:

- Tests where the test engineer passes to an assertion primitive to force the test to fail. Such assertion calls are intended to be executed only if something goes wrong, and not if the test passes.

Code Example:

```
@Test
public void testMethod() {
    MyClass obj = new MyClass();
    int expected = 5;
    int actual = obj.doSomething();
    assertTrue(actual > 0);
    assertFalse(actual > 10);
    fail("Something went wrong");
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- RTj: a Java framework for detecting and refactoring rotten green test cases
- Rotten green tests in Java, Pharo and Python

1.4.1.35 Missing Assertions

Definition:

- The subject includes behavior which is not asserted by the test, whether implicitly or explicitly.
- The test method consists of an empty block

Code Example:

```
public void testSomething() {
  // TODO
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- A workbook repository of example test smells and what to do about them
- JUnit Anti-patterns
- On the Maintenance of System User Interactive Tests
- Smells in System User Interactive Tests

1.4.1.36 Missing Test

Definition:

- The smell occurs when a test method does not include an assertion. This smell is also known as unknown test.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Understanding Testability and Test Smells

1.4.1.37 No Assertions

Definition:

- Tests that have no assertions and require the manual verification of log outputs

Also Known As:

- Assertionless Test

Code Example:

```
IResult result = format.execute();
System.out.println(result.size());
Iterator iter = result.iterator();

while (iter.hasNext()) {
  IResult r = (IResult) iter.next();
  System.out.println(r.getMessage());
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Bad tests, good tests
- Categorising Test Smells
- Improving Student Testing Practices through a Lightweight Checklist Intervention.
- Smells of Testing (signs your tests are bad)

1.4.1.38 No Traces Left

Definition:

- Tests that, despite failing, do not produce any logs or indications of what went wrong

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-patterns in test automation

1.4.1.39 On The Fly

Definition:

- The test calculates an expected results during its execution instead of relying on pre-computed values

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- On the Maintenance of System User Interactive Tests
- Smells in System User Interactive Tests

1.4.1.40 Over Exertion Assertion

Definition:

- Where the implementation of an assertion is heavy and in the body of the test, rather than in an assertion library

Code Example:

```
List results = service.getFoosInOrder();

int previousOrder = results.get(0).getOrdering();
for(int i=0; i= because they can sometimes be of same value
  if (!(ordering>=previousOrder)) {
    Assert.fail("Not in ascending order");
  }
  previousOrder = ordering;
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.4.1.41 Over-Checking

Definition:

- The test performs some assertions that are not relevant for its scope.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- On the Maintenance of System User Interactive Tests
- Smells in System User Interactive Tests
- Test design for automation: Anti-patterns

1.4.1.42 Premature Assertions

Definition:

- Intermingled with a test's set-up of the data and objects it depends on are assertions which attempt to ensure that set-up was successful. The net effect of this often resembles a "Arrange-Assert-Act-Assert" pattern.

Code Example:

```
# Subject under test
class Plane
  def initialize(air_traffic_control)
    @air_traffic_control = air_traffic_control
  end

  def take_off
    if started? &&
        on_airstrip? &&
        @air_traffic_control.ready_for_takeoff?(self)
      (rand * 1000).round
    else
      0
    end
  end
end


# Test
class PrematureAssertions < SmellTest
  def setup
    @air_traffic_control = AirTrafficControl.new
    @plane = Plane.new(@air_traffic_control)
    super
```

(continues on next page)

(continued from previous page)

```ruby
  end

  def test_take_off_when_ready
    @plane.start
    assert_equal true, @plane.started?
    @plane.taxi
    assert_equal true, @plane.on_airstrip?
    @air_traffic_control.approve(@plane)
    assert_equal true, @air_traffic_control.ready_for_takeoff?(@plane)

    altitude = @plane.take_off

    assert altitude > 0, "altitude should be greater than 0"
  end

  def test_does_not_take_off_when_not_ready
    altitude = @plane.take_off

    assert_equal 0, altitude
  end
end

# Fake production implementations to simplify example test of subject
class Plane
  def start
    @started = true
  end

  def started?
    @started if defined?(@started)
  end

  def taxi
    @taxied = true
  end

  def taxied?
    @taxied
  end

  def on_airstrip?
    taxied?
  end
end

class AirTrafficControl
  def approve(plane)
    @approved = true
  end

  def ready_for_takeoff?(plane)
    @approved
```

(continues on next page)

(continued from previous page)

```
  end
end
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- A workbook repository of example test smells and what to do about them

1.4.1.43 Primitive Assertion

Definition:

- Assertions that use primitive content to express intent

Code Example:

```
@Test
public void testMethod() {
    MyClass obj = new MyClass();
    int expected = 5;
    int actual = obj.doSomething();
    assertThat("Result of doSomething() should be greater than expected",
            actual, greaterThan(expected));
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Developer test anti-patterns by lasse koskela

1.4.1.44 Redundant Assertion

Definition:

- This smell occurs when test methods contain assertion statements that are either always true or always false. A test is intended to return a binary outcome of whether the intended result is correct or not, and should not return the same output regardless of the input.

Code Example:

```java
@Test
public void testTrue() {
    assertEquals(true, true);
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- An Exploratory Study on the Refactoring of Unit Test Files in Android Applications
- Automatic Identification of High-Impact Bug Report by Product and Test Code Quality
- Automatic generation of smell-free unit tests
- Categorising Test Smells
- Developers perception on the severity of test smells: an empirical study
- Handling Test Smells in Python: Results from a Mixed-Method Study
- Investigating Test Smells in JavaScript Test Code
- JUnit Anti-patterns
- On the Distribution of "Simple Stupid Bugs" in Unit Test Files: An Exploratory Study
- On the diffusion of test smells and their relationship with test code quality of Java projects
- On the distribution of test smells in open source Android applications: an exploratory study
- On the influence of Test Smells on Test Coverage
- On the test smells detection: an empirical study on the jnose test accuracy
- On the use of test smells for prediction of flaky tests
- PyNose: A Test Smell Detector For Python
- Software Unit Test Smells
- Test Smell Detection Tools: A Systematic Mapping Study
- The secret life of test smells-an empirical study on test smell evolution and maintenance
- What the Smell? An Empirical Investigation on the Distribution and Severity of Test Smells in Open Source Android Applications

- tsDetect: an open source test smells detection tool

1.4.1.45 Returning Assertion

Definition:

- A test method that has an assertion and returns a value.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Automatic generation of smell-free unit tests
- Rule-based Assessment of Test Quality
- Test Smell Detection Tools: A Systematic Mapping Study

1.4.1.46 Second Guess The Calculation

Definition:

- Where rather than using concrete test data, we use something that needs us to calculate the correct answer ahead of assertion

Code Example:

```java
@Test
public void testCalculatePrice() {
    Order order = new Order();
    order.addProduct(new Product("Product A", 10.0, 2));
    order.addProduct(new Product("Product B", 5.0, 3));

    double expectedPrice = 10.0 * 2 + 5.0 * 3;
    double actualPrice = order.calculatePrice();

    assertEquals(expectedPrice, actualPrice, 0.001);
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.4.1.47 Self-Test

Definition:

- Tests that do not compare a result with an expected value, but with the result itself

Code Example:

```java
@Test
public void shouldGetMethodsForPoland() {
  //given
  List<PaymentMethod> all = Lists.newArrayList(PaymentMethod.values());
  List<PaymentMethod> methodsAvailableInPoland = Lists.newArrayList();

  for (PaymentMethod method : all) {
    if (method.isEligibleForCountry("PL")) {
      methodsAvailableInPoland.add(method);
    }
  }

  //when
  List<PaymentMethod> methodsForCountry = PaymentMethod
    .getMethodsForCountry("PL", all);

  //then
  assertThat(methodsForCountry).isEqualTo(methodsAvailableInPoland);
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Bad tests, good tests

1.4.1.48 Sensitive Equality

Definition:

- Occurs when the toString method is used within a test method. Test methods verify objects by invoking the default toString() method of the object and comparing the output against an specific string. Changes to the implementation of toString() might result in failure. The correct approach is to implement a custom method within the object to perform this comparison.

Also Known As:

- The Butterfly

Code Example:

```
@Test
public void test1() throws UnknownHostException {

    String peersPacket = "F8 4E 11 F8 4B C5 36 81 " +
        "CC 0A 29 82 76 5F B8 40 D8 D6 0C 25 80 FA 79 5C " +
        "FC 03 13 EF DE BA 86 9D 21 94 E7 9E 7C B2 B5 22 " +
        "F7 82 FF A0 39 2C BB AB 8D 1B AC 30 12 08 B1 37 " +
        "E0 DE 49 98 33 4F 3B CF 73 FA 11 7E F2 13 F8 74 " +
        "17 08 9F EA F8 4C 21 B0";

    byte[] payload = Hex.decode(peersPacket);

    byte[] ip = decodeIP4Bytes(payload, 5);

    assertEquals(InetAddress.getByAddress(ip).toString(), ("/54.204.10.41"));
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- A preliminary evaluation on the relationship among architectural and test smells
- An Empirical Study into the Relationship Between Class Features and Test Smells
- An Exploratory Study on the Refactoring of Unit Test Files in Android Applications
- An empirical analysis of the distribution of unit test smells and their impact on software maintenance
- An empirical investigation into the nature of test smells
- An exploratory study of the relationship between software test smells and fault-proneness
- Analyzing Test Smells Refactoring from a Developers Perspective
- Are test smells really harmful? An empirical study
- Assessing diffusion and perception of test smells in scala projects

- Automatic Identification of High-Impact Bug Report by Product and Test Code Quality
- Automatic generation of smell-free unit tests
- Categorising Test Smells
- Characterizing High-Quality Test Methods: A First Empirical Study
- Developers perception on the severity of test smells: an empirical study
- Enhancing developers' awareness on test suites' quality with test smell summaries
- Handling Test Smells in Python: Results from a Mixed-Method Study
- Machine Learning-Based Test Smell Detection
- On the Distribution of "Simple Stupid Bugs" in Unit Test Files: An Exploratory Study
- On the Relation of Test Smells to Software Code Quality
- On the diffusion of test smells and their relationship with test code quality of Java projects
- On the diffusion of test smells in automatically generated test code: an empirical study
- On the distribution of test smells in open source Android applications: an exploratory study
- On the influence of Test Smells on Test Coverage
- On the interplay between software testing and evolution and its effect on program comprehension
- On the test smells detection: an empirical study on the jnose test accuracy
- On the use of test smells for prediction of flaky tests
- Refactoring Test Code
- Refactoring Test Smells: A Perspective from Open-Source Developers
- Revisiting Test Smells in Automatically Generated Tests: Limitations, Pitfalls, and Opportunities
- Scented since the beginning: On the diffuseness of test smells in automatically generated test code
- SoCRATES: Scala radar for test smells
- Software Unit Test Smells
- Test Smell Detection Tools: A Systematic Mapping Study
- TestQ: Exploring Structural and Maintenance Characteristics of Unit Test Suites
- The secret life of test smells-an empirical study on test smell evolution and maintenance
- To What Extent Can Code Quality be Improved by Eliminating Test Smells?
- Toward static test flakiness prediction: a feasibility study
- Towards Automated Tools for Detecting Test Smells: An Empirical Investigation into the Nature of Test Smells
- What the Smell? An Empirical Investigation on the Distribution and Severity of Test Smells in Open Source Android Applications
- Why do builds fail?—A conceptual replication study
- tsDetect: an open source test smells detection tool
- xUnit test patterns: Refactoring test code

1.4.1.49 Shotgun Surgery

Definition:

- Multiple places need to be modified with a single change.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Bad smells and refactoring methods for GUI test script

1.4.1.50 Testing The Internal Monologue

Definition:

- Where the writer of the test has been so focused on the lines of code in their implementation that they haven't sought ways to observe the behaviour of the system from the outside.

Code Example:

```
@Spy
private Service service = new Service( ... );

@Test
public void withInputAItDoesTheThing() {
    service.process("A");
    verify(service).internalCall();
}

@Test
public void withInputBItDoesntDoTheThing() {
    service.process("B");
    verify(service, never()).internalCall();
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.4.1.51 The Butterfly

Definition:

- You have to test something which contains data that changes all the time, like a structure which contains the current date, and there is no way to nail the result down to a fixed value. The ugly part is that you don't care about this value at all. It just makes your test more complicated without adding any value.

Also Known As:

- Sensitive Equality

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Unit testing Anti-patterns catalogue

1.4.1.52 The Nitpicker

Definition:

- A unit test which compares a complete output when it's really only interested in small parts of it, so the test has to continually be kept in line with otherwise unimportant details. Endemic in web application testing.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-Patterns - Digital Tapestry
- Test-Driven Development: TDD Anti-Patterns
- Unit Testing Anti-Patterns, Full List

1.4.1.53 Under-The-Carpet Assertion

Definition:

- Some assertions put into comments

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Rule-based Assessment of Test Quality

1.4.1.54 Under-The-Carpet Failing Assertion

Definition:

- A test having the smell Under-the-carpet failing Assertion is a test that returns a successful test-result, but contains hidden assertions. A hidden assertion is an assertion that is put into comments, is not executed when the test runs, and which would actually throw an Error or Failure if the comment were removed.

Also Known As:

- Commented Code in the Test

Code Example:

```
ICImporterTest >> #testImport
  ...
  self assert: eventAtDate textualDescription = 'blabla'"
  self assert: eventAtDate categories anyOne
  = (calendar categoryWithSummary: 'business').
  "self assert: ...
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Assessing test quality - TestLint
- Rule-based Assessment of Test Quality
- Test Smell Detection Tools: A Systematic Mapping Study
- Test Smell Detection Tools: A Systematic Mapping Study

1.4.1.55 Using Assertions As A Substitute For All Class-Based Exceptions

Definition:

- Though the line between when to use assertions instead of class-based exceptions can be blurry at times, I think a general rule of thumb here would be to use class-based exceptions in any situation where there is a chance of recovery.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Abap assertion anti-patterns

1.4.1.56 Using Assertions As A Substitute For All Defensive Programming Techniques

Definition:

- When developers get carried away with assertions, assuming that they are an across-the-board replacement for defensive programming techniques.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Abap assertion anti-patterns

1.4.1.57 Using The Wrong Assert

Definition:

- There are a large number of different methods beginning with assert defined in the Assert class. Each of these methods has slightly different arguments and semantics about what they are asserting. However, many programmers seem to stick with a single assertion method: assertTrue, and then force the argument of this method into the required boolean expression.

Code Example:

```
assertTrue("Objects must be the same", expected == actual);
assertTrue("Objects must be equal", expected.equals(actual));
assertTrue("Object must be null", actual == null);
assertTrue("Object must not be null", actual != null);
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- JUnit Anti-patterns

1.4.2 Issues in exception handling

1.4.2.1 Catching Unexpected Exceptions

Definition:

- When writing production code, developers are generally aware of the problems of uncaught exceptions, and so are relatively diligent about catching exceptions and logging the problem. In the case of unit tests, however, this pattern is completly wrong!

Code Example:

```
public void testCalculation() {
  try {
      deepThought.calculate();
      assertEquals("Calculation wrong", 42, deepThought.getResult());
  }
  catch(CalculationException ex) {
      Log.error("Calculation caused exception", ex);
  }
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Categorising Test Smells
- JUnit Anti-patterns

1.4.2.2 Exception Catch/Throw

Definition:

- Passing or failing of a test case depends on custom exception handling code or exception throwing, which may hide real problems and hamper debugging.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- The secret life of test smells-an empirical study on test smell evolution and maintenance

1.4.2.3 Exception Catching Throwing

Definition:

- Occurs when a test method is explicitly dependent on the production method throwing an exception

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- On the influence of Test Smells on Test Coverage

1.4.2.4 Exception Handling

Definition:

- This smell occurs when a test method explicitly a passing or failing of a test method is dependent on the production method throwing an exception. Developers should utilize JUnit's exception handling to automatically pass/fail the test instead of writing custom exception handling code or throwing an exception.

Code Example:

```
@Test
public void realCase() {
    Point p34 = new Point("34", 556506.667, 172513.91, 620.34, true);
    Point p45 = new Point("45", 556495.16, 172493.912, 623.37, true);
    Point p47 = new Point("47", 556612.21, 172489.274, 0.0, true);
    Abriss a = new Abriss(p34, false);
```

(continues on next page)

(continued from previous page)

```
    a.removeDAO(CalculationsDataSource.getInstance());
    a.getMeasures().add(new Measure(p45, 0.0, 91.6892, 23.277, 1.63));
    a.getMeasures().add(new Measure(p47, 281.3521, 100.0471, 108.384, 1.63));

    try {
        a.compute();
    } catch (CalculationException e) {
        Assert.fail(e.getMessage());
    }

    // test intermediate values with point 45
    Assert.assertEquals("233.2405",
        this.df4.format(a.getResults().get(0).getUnknownOrientation()));
    Assert.assertEquals("233.2435",
        this.df4.format(a.getResults().get(0).getOrientedDirection()));
    Assert.assertEquals("-0.1", this.df1.format(
        a.getResults().get(0).getErrTrans()));

    // test intermediate values with point 47
    Assert.assertEquals("233.2466",
        this.df4.format(a.getResults().get(1).getUnknownOrientation()));
    Assert.assertEquals("114.5956",
        this.df4.format(a.getResults().get(1).getOrientedDirection()));
    Assert.assertEquals("0.5", this.df1.format(
        a.getResults().get(1).getErrTrans()));

    // test final results
    Assert.assertEquals("233.2435", this.df4.format(a.getMean()));
    Assert.assertEquals("43", this.df0.format(a.getMSE()));
    Assert.assertEquals("30", this.df0.format(a.getMeanErrComp()));
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- An Exploratory Study on the Refactoring of Unit Test Files in Android Applications
- Automatic Identification of High-Impact Bug Report by Product and Test Code Quality
- Automatic generation of smell-free unit tests
- Characterizing High-Quality Test Methods: A First Empirical Study
- Handling Test Smells in Python: Results from a Mixed-Method Study
- Investigating Test Smells in JavaScript Test Code
- On the Distribution of “Simple Stupid Bugs” in Unit Test Files: An Exploratory Study

- On the diffusion of test smells and their relationship with test code quality of Java projects
- On the distribution of test smells in open source Android applications: an exploratory study
- On the influence of Test Smells on Test Coverage
- On the test smells detection: an empirical study on the jnose test accuracy
- PyNose: A Test Smell Detector For Python
- Pytest-Smell: a smell detection tool for Python unit tests
- Refactoring Test Smells With JUnit 5: Why Should Developers Keep Up-to-Date?
- Refactoring Test Smells: A Perspective from Open-Source Developers
- Software Unit Test Smells
- TEMPY: Test Smell Detector for Python
- Test Smell Detection Tools: A Systematic Mapping Study
- TestAXE: Automatically Refactoring Test Smells Using JUnit 5 Features
- The secret life of test smells-an empirical study on test smell evolution and maintenance
- Understanding Testability and Test Smells
- Understanding practitioners' strategies to handle test smells: a multi-method study
- What the Smell? An Empirical Investigation on the Distribution and Severity of Test Smells in Open Source Android Applications
- tsDetect: an open source test smells detection tool

1.4.2.5 Expected Exceptions And Verification

Definition:

- A verification happens after an expected exception is thrown and halts the test method execution

Code Example:

```
@Test(expected = RuntimeException.class)
public void shouldSaveFailureInformationWhenExceptionOccurWhenAddingDomain() {
  //given
  doThrow(new RuntimeException()).when(dnsService)
    .addDomainIfMissing(DOMAIN_ADDRESS);

  //when
  domainRegistrator.registerDomain(domain);

  //then
  verify(domainService)
    .saveDomain(domain, false, DNS_FAILURES + 1);
}
```

References:

Quality attributes

- - Code Example

- - Cause and Effect
- - Frequency
- - Refactoring

- Bad tests, good tests

1.4.2.6 Expecting Exceptions Anywhere

Definition:

- Tests that do not track the step that raised the expected exception and pass

Code Example:

```
@Test(expected=IndexOutOfBoundsException.class)
    public void testMyList() {
      MyList<Integer> list = new MyList<Integer>();
      list.add(1);
      list.add(2);
      list.add(3);
      list.add(3);
      list.add(4);
      assertTrue(4 == list.get(4));
      assertTrue(2 == list.get(1));
      assertTrue(3 == list.get(2));
      list.get(6);
  }
  @Test(expected=IndexOutOfBoundsException.class)
    public void testNegative() {
      MyList<Integer> list = new MyList<Integer>();
      list.add(1);
      list.add(2);
      list.add(3);
      list.add(3);
      list.add(4);
      list.get(-1);
  }
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Bad tests, good tests

1.4.2.7 Issues In Exception Handling

Definition:

- A test case that succeeds even when an exception is thrown. Mostly, the exception is ignored along with its type, which represents specific semantics. This may lead to false results, e.g. when testing the authentication protocol.

Also Known As:

- The Secret Catcher

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Did You Remember To Test Your Tokens?

1.4.2.8 No Error Handling In Tests

Definition:

- Avoid such error handling in tests, as this kind of err should be a tech handler but not a business matters.

Code Example:

- No code examples yet…

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Development Patterns and Anti-Patterns - Medium

1.4.2.9 The Greedy Catcher

Definition:

- A unit test which catches exceptions and swallows the stack trace, sometimes replacing it with a less informative failure message, but sometimes even just logging (c.f. Loudmouth) and letting the test pass.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-Patterns - Digital Tapestry
- Categorising Test Smells
- Test-Driven Development: TDD Anti-Patterns
- Unit Testing Anti-Patterns, Full List

1.4.2.10 The Secret Catcher

Definition:

- A test that at first glance appears to be doing no testing, due to absence of assertions. But "The devil is in the details".. the test is really relying on an exception to be thrown and expecting the testing framework to capture the exception and report it to the user as a failure.

Also Known As:

- The Silent Catcher, Issues in Exception Handling

Code Example:

```
[Test]
public void ShouldNotThrow()
{
  DoSomethingThatShouldNotThrowAnException();
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-Patterns - Digital Tapestry

- Categorising Test Smells
- Test-Driven Development: TDD Anti-Patterns
- Unit Testing Anti-Patterns, Full List
- Unit testing Anti-patterns catalogue

1.4.2.11 The Silent Catcher

Definition:

- A test that passes if an exception is thrown. Even if the exception that actually occurs is one that is different than the one the developer intended.

Also Known As:

- The Secret Catcher

Code Example:

```
[Test]
[ExpectedException(typeof(Exception))]
public void ItShouldThrowDivideByZeroException()
{
  // some code that throws another exception yet passes the test
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-Patterns - Digital Tapestry
- Smells in software test code: A survey of knowledge in industry and academia
- Unit testing Anti-patterns catalogue

1.4.3 Issues in setup

1.4.3.1 Badly Used Fixture

Definition:

- A Badly Used Fixture is a fixture that is not fully used by the tests in the test-suite.

Also Known As:

- General Fixture

Code Example:

```java
public class ShoppingCartTest {
    private ShoppingCart cart;

    @Before
    public void setUp() {
        cart = new ShoppingCart();
        cart.addItem(new Item("Shirt", 25.0));
        cart.addItem(new Item("Pants", 50.0));
    }

    @Test
    public void testEmptyCart() {
        cart.clear();
        assertTrue(cart.getItems().isEmpty());
    }

    @Test
    public void testAddItem() {
        cart.addItem(new Item("Shoes", 75.0));
        assertEquals(3, cart.getItems().size());
    }

    // This test doesn't need to use the pre-selected items
    @Test
    public void testRemoveItem() {
        cart.removeItem(new Item("Shirt", 25.0));
        assertEquals(1, cart.getItems().size());
    }
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Assessing test quality - TestLint

1.4.3.2 Bury The Lede

Definition:

- A test's setup is so onerous that the reader forgets the purpose of the subject by the time they reach the relevant invocation and assertion of the subject.

Code Example:

```
// Subject under test
function verifyCardholderZip (cardId, zip) {
  var card = repo.find(cardId)
  var user = repo.find(card.userId)
  var address = repo.find(user.addressId)

  return address.zip === zip
}

// Test
module.exports = {
  trueIfZipMatches: function () {
    var address = {street: '123 Sesame', city: 'Cbus', state: 'OH', zip: '42'}
    repo.save(address)
    var user = {
      addressId: address.id,
      name: 'Jane',
      age: 12,
      income: '$12.48'
    }
    repo.save(user)
    var issuer = {bankName: 'Bank Co'}
    repo.save(issuer)
    var card = {
      userId: user.id,
      apr: 17.8,
      number: '1234 0000 2828 4494',
      ccv: 364,
      issuerId: issuer.id
    }
    repo.save(card)

    var result = verifyCardholderZip(card.id, '42')

    assert.equal(result, true)
  }
}

// Fake production implementations to simplify example test of subject
var repo = {
  __items: {},
  nextId: 1,
  save: function (obj) {
    if (!obj.id) obj.id = repo.nextId++
```

(continues on next page)

(continued from previous page)

```
    // Gotcha!
    if (obj.zip) validateAddress(obj)
    if (obj.addressId) validateUser(obj)
    if (obj.userId) validateCard(obj)

    repo.__items[obj.id] = obj
  },
  find: function (id) {
    return repo.__items[id]
  }
}

function validateAddress (address) {
  requireProperties(address, ['street', 'city', 'state'])
}

function validateUser (user) {
  requireProperties(user, ['name', 'age', 'income'])
}

function validateCard (card) {
  requireProperties(card, ['apr', 'number', 'ccv'])
  requireRelation(card, 'issuerId', 'bankName')
}

function requireProperties (obj, props) {
  props.forEach(function (prop) {
    if (!obj.hasOwnProperty(prop)) {
      throw new Error('ERROR: "' + prop + '" required on ' + JSON.stringify(obj))
    }
  })
}

function requireRelation (obj, idKey, prop) {
  var relation = repo.find(obj[idKey])
  if (!relation || !relation.hasOwnProperty(prop)) {
    throw new Error('ERROR: "' + prop + '" required on "' + idKey + '" of ' +
      JSON.stringify(obj))
  }
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- A workbook repository of example test smells and what to do about them

1.4.3.3 Complicated Set Up Scenarios Within The Tests Themselves

Definition:

- Whilst there's a place for automated end-to-end scenarios (I call these user journies), I prefer most acceptance tests to jump straight to the point.

Code Example:

```
Scenario: Accept Visa and Mastercard for Australia
Given I am on the home page for Australia
And I choose the tea menu
And I select some 'green tea'
And I add the tea to my basket
And I choose to checkout
Then I should see 'visa' is accepted
And I should see 'mastercard' is accepted
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Five automated acceptance test anti-patterns

1.4.3.4 Curdled Test Fixtures

Definition:

- Where there's an inappropriate union of tests in the same fixture, or splitting into multiple fixtures where one would be better

Code Example:

```
class MyTest {
  // some test input or expected output
  private static final SomeObject COMPLEX_DATA = new ...;

  private Thing whatWeAreTesting = new ...;

  // ... other resources

  @BeforeEach
  void beforeEach() {
      // some additional setup
  }
  @AfterEach
  void afterEach() {
      // some tidy up
```

(continues on next page)

(continued from previous page)

```
  }

  @Test
  void testOne() { ... }
```

}

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.4.3.5 Empty Shared-Fixture

Definition:

- A test that defines a fixture with an empty body.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Automatic generation of smell-free unit tests
- Rule-based Assessment of Test Quality
- Test Smell Detection Tools: A Systematic Mapping Study

1.4.3.6 Excessive Inline Setup

Definition:

- Too much setup in every test means the reader has to skip most of the test to get to the meat.

References:

Quality attributes

- - Code Example

- - Cause and Effect
- - Frequency
- - Refactoring

- Categorising Test Smells
- Smells of Testing (signs your tests are bad)

1.4.3.7 Excessive Setup

Definition:

- Many dependencies you have to create beforehand (such as classes, operating system dependencies, databases - basically anything that removes the attention to the testing goal).

Also Known As:

- Large Setup Methods, Inappropriately Shared Fixture, The Mother Hen, The Stranger, The Distant Relative, The Cuckoo

Code Example:

```
jest.mock('compression')
jest.mock('connect')
jest.mock('serve-static')
jest.mock('serve-placeholder')
jest.mock('launch-editor-middleware')
jest.mock('@nuxt/utils')
jest.mock('@nuxt/vue-renderer')
jest.mock('../src/listener')
jest.mock('../src/context')
jest.mock('../src/jsdom')
jest.mock('../src/middleware/nuxt')
jest.mock('../src/middleware/error')
jest.mock('../src/middleware/timing')

describe('server: server', () => {
  const createNuxt = () => ({
    options: {
      dir: {
        static: 'var/nuxt/static'
      },
      srcDir: '/var/nuxt/src',
      buildDir: '/var/nuxt/build',
      globalName: 'test-global-name',
      globals: { id: 'test-globals' },
      build: {
        publicPath: '__nuxt_test'
      },
      router: {
        base: '/foo/'
      },
      render: {
        id: 'test-render',
```

(continues on next page)

(continued from previous page)

```
        dist: {
          id: 'test-render-dist'
        },
        static: {
          id: 'test-render-static',
          prefix: 'test-render-static-prefix'
        }
      },
      server: {},
      serverMiddleware: []
    },
    hook: jest.fn(),
    ready: jest.fn(),
    callHook: jest.fn(),
    resolver: {
      requireModule: jest.fn(),
      resolvePath: jest.fn().mockImplementation(p => p)
    }
  })

  beforeAll(() => {
    jest.spyOn(path, 'join').mockImplementation((...args) => `join(${args.join(', ')})`)
    jest.spyOn(path, 'resolve').mockImplementation((...args) => `resolve(${args.join(',
↪')})`)
    connect.mockReturnValue({ use: jest.fn() })
    serveStatic.mockImplementation(dir => ({ id: 'test-serve-static', dir }))
    nuxtMiddleware.mockImplementation(options => ({
      id: 'test-nuxt-middleware',
      ...options
    }))
    errorMiddleware.mockImplementation(options => ({
      id: 'test-error-middleware',
      ...options
    }))
    createTimingMiddleware.mockImplementation(options => ({
      id: 'test-timing-middleware',
      ...options
    }))
    launchMiddleware.mockImplementation(options => ({
      id: 'test-open-in-editor-middleware',
      ...options
    }))
    servePlaceholder.mockImplementation(options => ({
      key: 'test-serve-placeholder',
      ...options
    }))
  })
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-Patterns - Digital Tapestry
- Categorising Test Smells
- TDD anti patterns - Chapter 1
- TDD anti-patterns - the liar, excessive setup, the giant, slow poke
- Test-Driven Development: TDD Anti-Patterns
- Unit Testing Anti-Patterns, Full List
- Unit testing Anti-patterns catalogue

1.4.3.8 Factories That Contain Unnecessary Data

Definition:

- When factories define more data than necessary for a model to function

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Rails Testing Antipatterns: Fixtures and Factories

1.4.3.9 Fragile Fixture

Definition:

- When a Standard Fixture is modified to accommodate a new test, several other tests fail. This is an alias for either Data Sensitivity or Context Sensitivity.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Categorising Test Smells
- xUnit test patterns: Refactoring test code

1.4.3.10 Fragile Locators

Definition:

- Any slight change on the UI would fail the tests, and it will require some maintenance work to update the specific locator.

Code Example:

- No code examples yet…

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- How To Avoid Anti-Patterns In Cypress

1.4.3.11 General Fixture

Definition:

- Occurs when a test case fixture is too general and the test methods only access part of it. A test setup/fixture method that initializes fields that are not accessed by test methods indicates that the fixture is too generalized. A drawback of it being too general is that unnecessary work is being done when a test method is run.

Also Known As:

- Badly Used Fixture

Code Example:

```
public void testGetFlightsByFromAirport_OneOutboundFlight() throws Exception {
      setupStandardAirportsAndFlights();
      FlightDto outboundFlight = findOneOutboundFlight();
      // Exercise System
      List flightsAtOrigin = facade.getFlightsByOriginAirport(
                    outboundFlight.getOriginAirportId());
      // Verify Outcome
      assertOnly1FlightInDtoList( "Flights at origin", outboundFlight,
                                  flightsAtOrigin);
  }

  public void testGetFlightsByFromAirport_TwoOutboundFlights() throws Exception {
      setupStandardAirportsAndFlights();
      FlightDto[] outboundFlights = findTwoOutboundFlightsFromOneAirport();
```

(continues on next page)

(continued from previous page)

```
      // Exercise System
      List flightsAtOrigin = facade.getFlightsByOriginAirport(
                     outboundFlights[0].getOriginAirportId());
      // Verify Outcome
      assertExactly2FlightsInDtoList( "Flights at origin", outboundFlights,
                                      flightsAtOrigin);
  }
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- A preliminary evaluation on the relationship among architectural and test smells
- A survey on test practitioners' awareness of test smells
- An Empirical Study into the Relationship Between Class Features and Test Smells
- An Exploratory Study on the Refactoring of Unit Test Files in Android Applications
- An analysis of information needs to detect test smells
- An empirical investigation into the nature of test smells
- An exploratory study of the relationship between software test smells and fault-proneness
- Are test smells really harmful? An empirical study
- Assessing diffusion and perception of test smells in scala projects
- Automated Detection of Test Fixture Strategies and Smells
- Automatic Identification of High-Impact Bug Report by Product and Test Code Quality
- Automatic Test Smell Detection Using Information Retrieval Techniques
- Automatic generation of smell-free unit tests
- Categorising Test Smells
- Characterizing High-Quality Test Methods: A First Empirical Study
- Characterizing the Relative Significance of a Test Smell
- Detecting redundant unit tests for AspectJ programs
- Enhancing developers' awareness on test suites' quality with test smell summaries
- Handling Test Smells in Python: Results from a Mixed-Method Study
- How are test smells treated in the wild? A tale of two empirical studies
- Investigating Severity Thresholds for Test Smells
- Just-In-Time Test Smell Detection and Refactoring: The DARTS Project

- Let's not
- Obscure Test
- On The Detection of Test Smells: A Metrics-Based Approach for General Fixture and Eager Test
- On the Distribution of "Simple Stupid Bugs" in Unit Test Files: An Exploratory Study
- On the diffusion of test smells and their relationship with test code quality of Java projects
- On the distribution of test smells in open source Android applications: an exploratory study
- On the influence of Test Smells on Test Coverage
- On the interplay between software testing and evolution and its effect on program comprehension
- On the test smells detection: an empirical study on the jnose test accuracy
- On the use of test smells for prediction of flaky tests
- PyNose: A Test Smell Detector For Python
- Refactoring Test Code
- SoCRATES: Scala radar for test smells
- Software Unit Test Smells
- Strategies for avoiding text fixture smells during software evolution
- Test Smell Detection Tools: A Systematic Mapping Study
- TestQ: Exploring Structural and Maintenance Characteristics of Unit Test Suites
- The secret life of test smells-an empirical study on test smell evolution and maintenance
- Towards Automated Tools for Detecting Test Smells: An Empirical Investigation into the Nature of Test Smells
- What We Know About Smells in Software Test Code
- What the Smell? An Empirical Investigation on the Distribution and Severity of Test Smells in Open Source Android Applications
- Why do builds fail?—A conceptual replication study
- tsDetect: an open source test smells detection tool
- xUnit test patterns: Refactoring test code

1.4.3.12 Hidden Test Data

Definition:

- The data are not directly visible and understandable in the test but are hidden in the fixture code

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- On the Maintenance of System User Interactive Tests
- Smells in System User Interactive Tests

1.4.3.13 Idle Ptc

Definition:

- A PTC is created but never started. A PTC which is not started is of no use for the test case.

Code Example:

```
testcase exampleTestCase ( ) runs on MainComponentType system SystemType {
  //...
  var ParallelComponentType exampleComponent := ParallelComponentType.create;
  map(self: aPort, system: aPort) ;
  connect(self: anotherPort, exampleComponent: aPort ) ;
  // no start here...
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- An approach to quality engineering of TTCN-3 test specifications
- Pattern-based Smell Detection in TTCN-3 Test Suites
- Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites

1.4.3.14 Inappropriately Shared Fixture

Definition:

- Several test cases in the test fixture do not even use or need the setup / teardown. Partly due to developer inertia to create a new test fixture… easier to just add one more test case to the pile

Also Known As:

- Excessive Setup

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Categorising Test Smells
- Unit testing Anti-patterns catalogue

1.4.3.15 Irrelevant Information

Definition:

- The test is exposing a lot of irrelevant details about the fixture that distract the test reader from what really affects the behavior of the SUT.

Code Example:

```
public void testAddItemQuantity_severalQuantity_v10(){
    //   Setup Fixture
    Address billingAddress = createAddress( "1222 1st St SW", "Calgary", "Alberta", "T2N␣
↪2V2", "Canada");
    Address shippingAddress = createAddress( "1333 1st St SW", "Calgary", "Alberta",
↪"T2N 2V2", "Canada");
    Customer customer = createCustomer( 99, "John", "Doe", new BigDecimal("30"),
                        billingAddress, shippingAddress);
    Product product = createProduct( 88,"SomeWidget",new BigDecimal("19.99"));
    Invoice invoice = createInvoice(customer);
    // Exercise SUT
    invoice.addItemQuantity(product, 5);
    // Verify Outcome
    LineItem expected = new LineItem(invoice, product,5, new BigDecimal("30"), new␣
↪BigDecimal("69.96"));
    assertContainsExactlyOneLineItem(invoice, expected);
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Detecting redundant unit tests for AspectJ programs
- Obscure Test

1.4.3.16 Isolated Ptc

Definition:

- A PTC is created and started, but neither connected to another component nor mapped to the TSI. A PTC which is not connected or mapped is isolated from all other components, especially the MTC, and is of no use for the test.

Code Example:

```
testcase exampleTestCase() runs on MainComponentType system SystemType {
  // . . .
  var ParallelComponentType exampleComponent := ParallelComponentType.create;
  // no map or connect statement shere!
  exampleComponent.start(exampleBehavior())
  exampleComponent.done
  // . . .
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Pattern-based Smell Detection in TTCN-3 Test Suites
- Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites

1.4.3.17 Large Fixture

Definition:

- Provides a large amount of data to the tests, making it difficult to understand the state of a unit under test and also obfuscating the purpose of the tests. Furthermore setup and teardown require a large amount of time slowing down the execution of the tests

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Rule-based Assessment of Test Quality

1.4.3.18 Max Instance Variables

Definition:

- A test method that has a large fixture

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Rule-based Assessment of Test Quality
- Test Smell Detection Tools: A Systematic Mapping Study

1.4.3.19 Noisy Logging

Definition:

- The test logs the state of the fixtures

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- On the Maintenance of System User Interactive Tests
- Smells in System User Interactive Tests

1.4.3.20 Noisy Setup

Definition:

- When a verbose sequence of low-level records that is difficult to comprehend is displayed in the setup

Code Example:

```
let(:product_1) { create(:product, name: 'iPad') }
let(:product_sale_1) { create(:product_sale, retail_price: 500, product: product_1) }
let(:product_2) { create(:product, name: 'iPhone') }
let(:product_sale_2) { create(:product_sale, retail_price: 500, product: product_2) }
let(:product_sales) { [product_sale_1, product_sale_2] }
let(:sale) { create(:sale, name: 'Apple Bundle', product_sales: product_sales) }
```

(continues on next page)

(continued from previous page)

```ruby
let(:user) { create(:user, name: 'Thiago') }
let!(:line_item) { create(:order_line_item, order: order, sale: sale) }
let(:order) { create(:order, user: user) }

it 'retrieves the expected data' do
  # Run the query and make assertions
end
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Rails Testing Antipatterns: Fixtures and Factories

1.4.3.21 Obscure In-Line Setup

Definition:

- An in-line setup should consist of only the steps and values essential to understanding the test. Essential but irrelevant steps should be encapsulated into helper methods. An obscure in-line setup covers too much setup functionality within the test method. This can hinder developers in seeing the relevant verification steps of the test.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Automated Detection of Test Fixture Strategies and Smells
- Automatic generation of smell-free unit tests
- PyNose: A Test Smell Detector For Python
- Strategies for avoiding text fixture smells during software evolution
- Test Smell Detection Tools: A Systematic Mapping Study

1.4.3.22 Oversharing On Setup

Definition:

- Where every test sets up a lot of shared data which only some tests need.

Code Example:

```
beforeEach(() => {
  databaseConnection = openDatabase();
  inputFile = loadBigFile();
  userList = loadUserList();
  imageData = loadImageBytes();
});
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.4.3.23 Refused Bequest

Definition:

- A mega setup that some tests ignore, some use, and most rely on but only for a few lines. When any individual test fails it's hard to parse the setup to see what's going on.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Categorising Test Smells
- Improving Student Testing Practices through a Lightweight Checklist Intervention.
- Smells of Testing (signs your tests are bad)

1.4.3.24 Scattered Test Fixtures

Definition:

- Test fixtures are defined in test cases to set up the environment for test execution. A base test case may define a general test fixture (i.e., a method annotated by @BeforeClass or @Before). When a child test case inherits a base test case, the child test case may define its own test fixture, but may also call the test fixture of the base test case. In multi-level inheritance, when each child test case has its own test fixture, the test fixture code becomes scattered and difficult to maintain.

Code Example:

- No code examples yet…

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Revisiting Test Impact Analysis in Continuous Testing From the Perspective of Code Dependencies

1.4.3.25 Separate Fixture Files

Definition:

- It's now a bit hard to tell what the data look like in the test code. If the data isn't too large, especially when I need only a small fraction of the data. I would define an in-file variable to demonstrate what we expect, and then use it in the test.

Code Example:

- No code examples yet…

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Unveiling 6 Anti-Patterns in React Test Code: Pitfalls to Avoid

1.4.3.26 Share The World

Definition:

- where the test sets up all its resources at the start of the fixture, leading to either bleeding of state between tests, or extra work to keep things tidy

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.4.3.27 Superfluous Setup Data

Definition:

- Occurs when testing queries or filters, in which you only expect to get a subset of the data back. The underlying idea is that, in order to be thorough, "extra" data should be present to show that the query or filter works as required.

Code Example:

```
@Test
public void givenMultipleWidgetsExistWhenQueriedByNameThenOnlyWidgetAFound() {
insertDefaultWidget("a");
insertDefaultWidget("b");
insertDefaultWidget("c");

WidgetQuery widgetQuery = new WidgetQuery();
List<Widget> results = widgetQuery.findByName("a");

assertEquals(1, results.size());
assertEquals("a", results[0].getName());
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Testing anti-patterns

1.4.3.28 Taking Environment State For Granted

Definition:

- Taking environment state for granted is often over optimistic. Unlike in unit tests, end-to-end setup gives little control over test environment state, particularly when it is shared with other teams. When the test starts failing it might be because of a new bug introduced or because environment is not in a state your test needs.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-patterns in test automation

1.4.3.29 Test Maverick

Definition:

- A test method is a maverick when the class comprising the test method contains an implicit setup, but the test method is completely independent from the implicit setup procedure. The setup procedure will be executed before the test method is executed, but it is not needed. Also, understanding the effect-cause relationship between setup and test method can be hampered. Discovering that test methods are unrelated from the implicit setup can be time consuming.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Automated Detection of Test Fixture Strategies and Smells
- Automatic generation of smell-free unit tests
- PyNose: A Test Smell Detector For Python
- Strategies for avoiding text fixture smells during software evolution
- Test Smell Detection Tools: A Systematic Mapping Study

1.4.3.30 Test Objects Initialized In A Describe Block

Definition:

- When tests share the same reference to an object, meaning that any single test can mutate the object for following tests.

Code Example:

- No code examples yet…

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- JavaScript Testing Anti-Patterns

1.4.3.31 Test Objects Initialized In Each Test

Definition:

- When the test object is initialized in each test case

Code Example:

- No code examples yet…

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- JavaScript Testing Anti-Patterns

1.4.3.32 Test Setup Is Somewhere Else

Definition:

- Where the test method just does the assertions, not the given/when part; this can be acceptable in the case of several tests on a single shared expensive resource setup, but seldom is at other times.

Code Example:

```
@Test
void theOperationIsSuccessful() {
    assertTrue(service.isLastOperationSuccessful());
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.4.3.33 The Cuckoo

Definition:

- A unit test which sits in a test case with several others, and enjoys the same (potentially lengthy) setup process as the other tests in the test case, but then discards some or all of the artifacts from the setup and creates its own.

Also Known As:

- Excessive Setup

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-Patterns - Digital Tapestry
- Categorising Test Smells
- Unit Testing Anti-Patterns, Full List
- Unit testing Anti-patterns catalogue

1.4.3.34 The Mother Hen

Definition:

- A common setup which does far more than the actual test cases need. For example creating all sorts of complex data structures populated with apparently important and unique values when the tests only assert for presence or absence of something.

Also Known As:

- Excessive Setup

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Categorising Test Smells
- Unit testing Anti-patterns catalogue

1.4.3.35 There Is Too Much Setup To Run The Test Cases

Definition:

- Tests require an excessive amount of setup just to be able to run. This may be in the Arrange stage of the individual tests or in a setup method that runs before the tests. There can be hundreds of lines of code in some cases where there is too much setup needed. This makes it almost impossible to understand what is being tested because of all the extra in the setup.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-Patterns In Unit Testing

1.4.3.36 Unused Definition

Definition:

- Unused code should be removed. Note that only local definitions can be removed safely because they cannot be accessed from outside the defining unit. For global definitions there might exist references in modules which have not been considered.

Code Example:

```
function unused_f ( in integer i ) return integer {
  var integer j := 4 2 , k ;
  return i + j ;
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- An approach to quality engineering of TTCN-3 test specifications
- Pattern-based Smell Detection in TTCN-3 Test Suites
- Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites

1.4.3.37 Using Fixtures

Definition:

- When a test uses fixtures to prepare and reuse test data.

Code Example:

```
# spec/fixtures/users.yml
marko:
  first_name: Marko
  last_name: Anastasov
  phone: 555-123-6788
```

```
RSpec.describe User do
  fixtures :all

  describe "#full_name" do
    it "is composed of first and last name" do
      user = users(:marko)
      expect(user.full_name).to eql "Marko Anastasov"
    end
  end
end
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Rails Testing Antipatterns: Fixtures and Factories

1.4.3.38 Vague Header Setup

Definition:

- A vague header setup smell occurs when fields are solely initialized in the header of a class. We consider this a smell as the behavior of the code is not explicitly defined and depends on the field modifier (static or member) as well as on the implementation of the test framework. Vague header setups might hamper code comprehension and maintainability, as fields can be placed anywhere in the class. Further, in many test frameworks exception messages are more expressive for fields initialized in the setup.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Automated Detection of Test Fixture Strategies and Smells
- Automatic generation of smell-free unit tests
- Strategies for avoiding text fixture smells during software evolution
- Test Smell Detection Tools: A Systematic Mapping Study

1.4.3.39 Verifying In Setup Method

Definition:

- When there is an assertion method within the SetUp method.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency

- - Refactoring

- Handling Test Smells in Python: Results from a Mixed-Method Study
- TEMPY: Test Smell Detector for Python

1.4.3.40 Where Does This One Go?

Definition:

- similar to the Curdled Test Fixture, this is caused by many tests having the same entry point into the software, even though they represent different use cases. It can be a symptom of using integration-level tests to test low-level things, or weakly defining the rules for each test fixture.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.4.4 Issues in teardown

1.4.4.1 Activation Asymmetry

Definition:

- A default activation has no matching subsequent deactivation in the same statement block, or a deactivation has no matching previous activation.

Code Example:

```
altstep myAltstep (timer t) runs on MyComponent {
  [] any port.receive {
    setverdict(fail)
    log ("unexpected message")
  }
  [] t.timeout{
    setverdict(fail)
    log("timeout")
  }
}

function activateDefault(timer t) return default {
  // no deactivation in this function!
  return act ivate ( myAltstep ( t ) )
}
```

(continues on next page)

(continued from previous page)

```
 testcase myTestcase1 ( ) runs on MyComponent {
   timer t
   var default myDefaultVar := activate(myAltstep(t))
   t.start(10.0)
   alt{
     [] p.receive(charstring:("foo1")){
       p.send("ack")
     }
     [] p.receive(charstring:("bar1")) {
       p.send("nack")
     }
   }
   deactivate(myDefaultVar)
 }

testcase myTestcase2 () runs on MyComponent {
 timer t;
 // activation in function call
 var default myDefaultVar := activateDefault(t)
   t.start(10.0)
   alt {
     []p.receive(charstring:("foo2")) {
       p.send("ack")
     }
     []p.receive(charstring:("bar2")) {
       p.send("nack")
     }
   }
   deactivate (myDefaultVar)
 }

 testcase myTestcase3 ( ) runs on MyComponent {
   // deactivation in different statement blocks
   timer t
   var default myDefaultVar
   myDefaultVar := activate(myAltstep(t))
   t.start(10.0)

   if(2 > 1) {
     alt {
       [] p.receive(charstring:("foo5")){
         p.send("ack")
       }[] p.receive(charstring:("bar5")){
         p.send("nack")
       }
     }
     deactivate ( myDefaultVar )
   }
 }
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- An approach to quality engineering of TTCN-3 test specifications
- Pattern-based Smell Detection in TTCN-3 Test Suites
- Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites

1.4.4.2 Complex Teardown

Definition:

- Complex fixture teardown code is more likely to leave test environment corrupted by not cleaning up correctly. It is hard to verify that it has been written correctly and can easily result in “data leaks” that may later cause this or other tests to fail for no apparent reason.

Code Example:

```
public void testGetFlightsByOrigin_NoInboudFlight_SMRTD() throws exception{
  BigDecimal outboundAirport = createTestAirport("1OF");
  BigDecimal inboundAirport = null;
  FlightDto expFlightDto = null;

  try {
    inboundAirport = createTestAirport("1IF");
    expFlightDto =
          createTestFlight(outboundAirport, inboundAirport);

    List flightsAtDestination1 =
          facade.getFlightsByOriginAirport(inboundAirport);

    assertEquals(0, flightsAtDestination1.size());
  } finally {
    try {
      facade.removeFlight(expFlightDto.getFlightNumber());
    } finally {
      try {
        facade.removeAirport(inboundAirport);
      }finally {
        facade.removeAirport(outboundAirport);
      }
    }
  }
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- xUnit test patterns: Refactoring test code

1.4.4.3 External Shared-State Corruption

Definition:

- Integration tests with shared resources and no rollback

Code Example:

```
Similar to shared-state corruption.
Tests touch shared resources (either in memory or in external resources,such as␣
↪databases and filesystems)
without cleaning up or rolling back any changes they make to those resources.
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Chapter 8. The pillars of good unit tests

1.4.4.4 Generous Leftovers

Definition:

- An instance where one unit test creates data that is persisted somewhere, and another test reuses the data for its own devious purposes. If the "generator" is ran afterward, or not at all, the test using that data will outright fail.

Also Known As:

- Wet Floor, Sloppy worker

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-Patterns - Digital Tapestry
- Categorising Test Smells
- Test-Driven Development: TDD Anti-Patterns
- Unit Testing Anti-Patterns, Full List

1.4.4.5 Improper Clean Up After Tests Have Been Run

Definition:

- Improper cleanup occurs when the code that cleans up the mocks and anything created in the test is insufficient or entirely lacking. It may leave files open or objects in memory causing memory leaks. This can be especially important if your tests are doing any type of file manipulation or you are creating files specifically for testing.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-Patterns In Unit Testing

1.4.4.6 It Was Like That When I Got Here

Definition:

- a test fixture that doesn't take any care over managiong the pre and post-test state, leading to all manner of side effects, including The Leaky Cauldron and The Soloist

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.4.4.7 Leaving Temporary Files Behind

Definition:

- File is not removed

Code Example:

- No code examples yet…

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Development Patterns and Anti-Patterns - Medium

1.4.4.8 Not Idempotent

Definition:

- Fancy words meaning your test changes the state of something permanently and thereby introduces the possibility of order dependent tests.

Also Known As:

- Interacting Test With High Dependency

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Categorising Test Smells
- Smells of Testing (signs your tests are bad)

1.4.4.9 Shared-State Corruption

Definition:

- Tests sharing in-memory state without rolling back

Code Example:

```
public class SharedStateCorruption
{
  // shared person state
  Person person = new Person();

  public void CreateAnalyzer_GooFileName_ReturnsTrue()
  {
    // changes shared state
    person.AddNumber("055-4556684(34)");
    string found = person.FindPhoneStartingWith("055");
    Assert.AreEqual("055-4556684(34)", found)
  }

  public void FindPhoneStartingWith_NoNumbers_ReturnNull()
  {
    // reads shared state
    string found = person.FindPhoneStartingWith("0");
    Assert.IsNull(found);
  }
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Chapter 8. The pillars of good unit tests

1.4.4.10 Sloppy Worker

Definition:

- The test generates data but it omits dispensing with the data after finishing. This will cause itself and other tests to fail on consecutive runs.

Also Known As:

- Generous Leftovers

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Categorising Test Smells

1.4.4.11 Teardown Only Test

Definition:

- Test-suite only de ning teardown (unusual for unit tests)

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Automatic generation of smell-free unit tests
- Rule-based Assessment of Test Quality
- Test Smell Detection Tools: A Systematic Mapping Study

1.4.4.12 The Painful Clean-Up

Definition:

- where every test needs to build or clean up an expensive resource, like a database, as the separation of tests is weak, or the test is too large

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.4.4.13 The Soloist

Definition:

- a test that can only be run successfully when no other tests are running

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.4.4.14 Unrepeatable Test

Definition:

- A test behaves differently the first time it is run than how it behaves on subsequent test runs.

Code Example:

```
Suite.run()--> Green
Suite.run()--> Test C fails
Suite.run()--> Test C fails
# User resets something
Suite.run()--> Green
Suite.run()--> Test C fails
Suite.run()--> Test C fails
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Categorising Test Smells
- xUnit test patterns: Refactoring test code

1.4.4.15 Wet Floor

Definition:

- the test creates data that is persisted somewhere, but the test does not clean up when fin- ished. This causes tests (the same test, or possibly other tests) to fail on subsequent test runs

Also Known As:

- Generous leftovers

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-Patterns - Digital Tapestry
- Smells in software test code: A survey of knowledge in industry and academia
- Unit testing Anti-patterns catalogue

1.5 Test execution - behavior

1.5.1 Other test execution - behavior

1.5.1.1 Abnormal Utf-Use

Definition:

- Test-suite overriding the default behavior of the unit testing framework

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Automatic generation of smell-free unit tests
- Rule-based Assessment of Test Quality
- Test Smell Detection Tools: A Systematic Mapping Study

1.5.1.2 Chatty Logging

Definition:

- Often a substitute for self-explanatory assertions or well defined test names, the test writes lots of data to the console or logs in order to explain test failures outside of the assertions

Code Example:

```
public class LoginTest {
    private WebDriver driver;

    @Before
    public void setUp() {
        driver = new ChromeDriver();
        driver.manage().window().maximize();
    }

    @After
    public void tearDown() {
        driver.quit();
    }

    @Test
    public void testValidLogin() {
        LoginPage loginPage = new LoginPage(driver);
        loginPage.enterUsername("testuser");
        loginPage.enterPassword("password");
        loginPage.clickLoginButton();

        String expectedUrl = "https://example.com/dashboard";
        String actualUrl = driver.getCurrentUrl();

        if (actualUrl.equals(expectedUrl)) {
            System.out.println("Login test passed.");
        } else {
            System.out.println("Login test failed. Expected URL: " + expectedUrl + ",␣
↪Actual URL: " + actualUrl);
        }

        assertTrue(actualUrl.equals(expectedUrl));
    }
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.5.1.3 Frequent Debugging

Definition:

- Indicates when ambiguity in test results require developers to conduct additional, manual debugging to identify the cause of the failed test.

Also Known As:

- Manual Debugging

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Automatic generation of smell-free unit tests
- Categorising Test Smells
- Inspecting Automated Test Code: A Preliminary Study
- Unit Test Smells and Accuracy of Software Engineering Student Test Suites
- xUnit test patterns: Refactoring test code

1.5.1.4 Interactive Test

Definition:

- An Interactive Test is a test that interrupts the automatic execution of a test, requiring manual actions from the user, for example pressing a button, closing a window or entering some data.

Also Known As:

- Manual Intervention

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Assessing test quality - TestLint

1.5.1.5 Manual Assertions

Definition:

- Manual assertions describes the scenarios where the JUnit assert or fail methods are ignored in favour of other methods confirming program accuracy.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- JUnit Anti-patterns

1.5.1.6 Manual Event Injection

Definition:

- A person must intervene during test execution to perform some manual action before the test can proceed.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Categorising Test Smells
- xUnit test patterns: Refactoring test code

1.5.1.7 Manual Fixture Setup

Definition:

- A person has to set up the test environment manually before the automated tests can be run. This may take the form of configuring servers, starting server processes or running scripts to set up a Prebuilt Fixture

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency

- - Refactoring

- Categorising Test Smells
- xUnit test patterns: Refactoring test code

1.5.1.8 Manual Intervention

Definition:

- A test case requires manual changes before the test is run, otherwise the test fails

Also Known As:

- Interactive Test, Manual Testing, Manual Test

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Automatic generation of smell-free unit tests
- Categorising Test Smells
- Inspecting Automated Test Code: A Preliminary Study
- xUnit test patterns: Refactoring test code

1.5.1.9 Manual Result Verification

Definition:

- We can run the tests but they almost always pass even when we know that the SUT is not returning the correct results.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Categorising Test Smells
- xUnit test patterns: Refactoring test code

1.5.1.10 Print Statement

Definition:

- Print statements in unit tests are redundant as unit tests are executed as part of an automated script and do not affect the failing or passing of test cases. Furthermore, they can increase execution time if the developer calls a long-running method from within the print method (i.e., as a parameter).

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Automatic Identification of High-Impact Bug Report by Product and Test Code Quality
- On the diffusion of test smells and their relationship with test code quality of Java projects
- On the influence of Test Smells on Test Coverage
- On the test smells detection: an empirical study on the jnose test accuracy
- The secret life of test smells-an empirical study on test smell evolution and maintenance

1.5.1.11 Redundant Print

Definition:

- Print statements in unit tests are redundant as unit tests are executed as part of an automated. Furthermore, they can consume computing resources or increase execution time if the developer calls an intensive/long-running method from within the print method (i.e., as a parameter).

Code Example:

```java
@Test
public void testTransform10mNEUAndBack() {
    Leg northEastAndUp10M = new Leg(10, 45, 45);
    Coord3D result = transformer.transform(Coord3D.ORIGIN, northEastAndUp10M);
    System.out.println("result = " + result);
    Leg reverse = new Leg(10, 225, -45);
    result = transformer.transform(result, reverse);
    assertEquals(Coord3D.ORIGIN, result);
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency

- - Refactoring

- A survey on test practitioners' awareness of test smells
- An Exploratory Study on the Refactoring of Unit Test Files in Android Applications
- Automatic Identification of High-Impact Bug Report by Product and Test Code Quality
- Automatic generation of smell-free unit tests
- Handling Test Smells in Python: Results from a Mixed-Method Study
- How are test smells treated in the wild? A tale of two empirical studies
- Investigating Test Smells in JavaScript Test Code
- On the Distribution of "Simple Stupid Bugs" in Unit Test Files: An Exploratory Study
- On the distribution of test smells in open source Android applications: an exploratory study
- On the use of test smells for prediction of flaky tests
- PyNose: A Test Smell Detector For Python
- Pytest-Smell: a smell detection tool for Python unit tests
- Software Unit Test Smells
- TEMPY: Test Smell Detector for Python
- Test Smell Detection Tools: A Systematic Mapping Study
- What the Smell? An Empirical Investigation on the Distribution and Severity of Test Smells in Open Source Android Applications
- tsDetect: an open source test smells detection tool

1.5.1.12 Requires Supervision

Definition:

- A test is not automatic – it needs user input to function. Someone has to watch the test and enter data periodically.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Smells of Testing (signs your tests are bad)

1.5.1.13 The Loudmouth

Definition:

- A unit test (or test suite) that clutters up the console with diagnostic messages, logging messages, and other miscellaneous chatter, even when tests are passing. Sometimes during test creation there was a desire to manually see output, but even though it's no longer needed, it was left behind.

Also Known As:

- Transcripting Test

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-Patterns - Digital Tapestry
- Categorising Test Smells
- Test-Driven Development: TDD Anti-Patterns
- Unit Testing Anti-Patterns, Full List

1.5.1.14 Transcripting Test

Definition:

- A Transcripting Test is writing information to the console or a global stream, for example the Transcript in Smalltalk, while it is running.

Also Known As:

- The Loudmouth

Code Example:

```
HeapTest >> #testExamples
  self shouldnt: [ self heapExample ] raise: Error.
  self shouldnt: [ self heapSortExample ] raise: Error.

HeapTest >> #heapSortExample
  "HeapTest new heapSortExample"
  "Sort a random collection of Floats and compare the results with ... ''
  | n rnd array time sorted |
  n := 10000. "# of elements to sort"
  rnd := Random new.

  Transcript cr; show:'Time for heap-sort: ', time printString,' msecs'.
  "The quicksort version"
```

(continues on next page)

(continued from previous page)

```
Transcript cr; show:'Time for quick-sort: ', time printString,' msecs'.
"The merge-sort version"
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Assessing test quality - TestLint
- Rule-based Assessment of Test Quality
- Test Smell Detection Tools: A Systematic Mapping Study

1.5.1.15 Trying To Use Ui Automation To Replace Manual Testing

Definition:

- The use of automation can drastically enhance the testing process. If used correctly, automation can reduce, not replace the manual testing resources required.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Top 17 Automated Testing Best Practices (Supported By Data)

1.5.1.16 Unnecessary Navigation

Definition:

- When the test has actions, taken for granted, not related to the things we want to check

Code Example:

```
public void CheckBookTitleDescriptionAndPrice_Version1()
{
    foreach (Book currentBook in testFile)
    {
        // Open the homepage
        selenium.Open("/HomePage.html");
```

(continues on next page)

(continued from previous page)

```
        selenium.WaitForPageToLoad("30000");

        // Login
        selenium.Click("link=Login");
        selenium.WaitForPageToLoad("30000");
        selenium.Type("TxtUserName", "TestAccount1");
        selenium.Type("TxtPassword", "TestPassword1");
        selenium.Click("Submit");
        selenium.WaitForPageToLoad("30000");

        // Search for the book we want to check
        selenium.Click("link=Search");
        selenium.WaitForPageToLoad("30000");
        selenium.Type("TxtSearchTerm", currentBook.Title);
        selenium.Click("Submit");
        selenium.WaitForPageToLoad("30000");

      // Open the Detailed Information page for the book we want to check<br />
        selenium.Click(currentBook.ID);
        selenium.WaitForPageToLoad("30000");

        // Check that the Detailed Information page has loaded
        Assert.AreEqual("Detailed Information page", selenium.GetTitle());

        // Compare the information on the Detailed Information page to our expected␣
↪results
        string actualBookTitle = selenium.GetText("BookTitle" + currentItem.ID);
        Assert.AreEqual(actualBookTitle, currentBook.Title);

        string actualBookDescription = selenium.GetText("BookDescription" + currentItem.
↪ID);
        Assert.AreEqual(actualBookDescription, currentBook.Description);

        string actualBookPrice = selenium.GetText("BookPrice" + currentItem.ID);
        Assert.AreEqual(actualBookPrice, currentBook.Price);
    }
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- How test automation with selenium can fail

1.5.2 Performance

1.5.2.1 Asynchronous Test

Definition:

- A few tests take inordinately long to run; those tests contain explicit delays.

Code Example:

```
public class RequestHandlerThreadTest extends TestCase {
  private static final int TWO_SECONDS = 3000;
  public void testWasInitialized_Async() throws InterruptedException {
    RequestHandlerThread sut = new RequestHandlerThread();

    sut.start();

    Thread.sleep(TWO_SECONDS);
    assertTrue(sut.initializedSuccessfully());
  }

  public void testHandleOneRequest_Async() throws InterruptedException {
    RequestHandlerThread sut = new RequestHandlerThread();
    sut.start();

    enqueueRequest(makeSimpleRequest());

    Thread.sleep(TWO_SECONDS);
    assertEquals(1, sut.getNumberOfRequestsCompleted());
    assertResponseEquals(makeSimpleResponse(), getResponse());
  }
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- xUnit test patterns: Refactoring test code

1.5.2.2 Factories Pulling Too Many Dependencies

Definition:

- Calling one factory may silently create many associated records, which accumulates to make the whole test suite slow (more on that later)

Code Example:

```
FactoryBot.define do
  factory :comment do
    post
    body "groundbreaking insight"
  end
end
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Rails Testing Antipatterns: Fixtures and Factories

1.5.2.3 Hardcoded Sleep

Definition:

- Has to wait 30s for each test run

Code Example:

- No code examples yet…

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Development Patterns and Anti-Patterns - Medium

1.5.2.4 Inefficient Waits

Definition:

- Adding a hard-coded wait that tells the test to pause for x number of seconds is a code smell.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Writing good gherkin

1.5.2.5 Long Running Tests

Definition:

- When the feedback loop of code, test, code gets too long it can encourage bad behavior.

Also Known As:

- Slow Test

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Smells of Testing (signs your tests are bad)

1.5.2.6 Sleeping For Arbitrary Amount Of Time

Definition:

- When a test becomes fragile to network or processing congestion

Code Example:

```
Thread.sleep(4*1000);
```

References:

Quality attributes

- - Code Example

- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-patterns in test automation

1.5.2.7 Sleepy Test

Definition:

- Developers introduce this smell when they need to pause execution of statements in a test method for a certain duration (i.e., simulate an external event) and then continuing with execution. Explicitly causing a thread to sleep can lead to unexpected results as the processing time for a task differs when executed in various environments and configurations.

Code Example:

```
public void testEdictExternSearch() throws Exception {
    final Intent i = new Intent(getInstrumentation().getContext(), ResultActivity.class);
    i.setAction(ResultActivity.EDICT_ACTION_INTERCEPT);
    i.putExtra(ResultActivity.EDICT_INTENTKEY_KANJIS, "");
    tester.startActivity(i);
    assertTrue(tester.getText(R.id.textSelectedDictionary).contains("Default"));
    final ListView lv = getActivity().getListView();
    assertEquals(1, lv.getCount());
    DictEntry entry = (DictEntry) lv.getItemAtPosition(0);
    assertEquals("Searching", entry.english);
    Thread.sleep(500);
    final Intent i2 = getStartedActivityIntent();
    final List result = (List) i2.getSerializableExtra(ResultActivity.INTENTKEY_RESULT_
↪LIST);
    entry = result.get(0);
    assertEquals("(adj-na,n,adj-no) blank space/vacuum/space/null (NUL)/(P)", entry.
↪english);
    assertEquals("", entry.getJapanese());
    assertEquals("", entry.reading);
    assertEquals(1, result.size());
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- An Exploratory Study on the Refactoring of Unit Test Files in Android Applications
- Automatic Identification of High-Impact Bug Report by Product and Test Code Quality

- Automatic generation of smell-free unit tests
- Characterizing High-Quality Test Methods: A First Empirical Study
- Handling Test Smells in Python: Results from a Mixed-Method Study
- Investigating Severity Thresholds for Test Smells
- Investigating Test Smells in JavaScript Test Code
- On the Distribution of "Simple Stupid Bugs" in Unit Test Files: An Exploratory Study
- On the diffusion of test smells and their relationship with test code quality of Java projects
- On the distribution of test smells in open source Android applications: an exploratory study
- On the influence of Test Smells on Test Coverage
- On the test smells detection: an empirical study on the jnose test accuracy
- On the use of test smells for prediction of flaky tests
- PyNose: A Test Smell Detector For Python
- Pytest-Smell: a smell detection tool for Python unit tests
- Software Unit Test Smells
- TEMPY: Test Smell Detector for Python
- Test Smell Detection Tools: A Systematic Mapping Study
- The secret life of test smells-an empirical study on test smell evolution and maintenance
- Understanding practitioners' strategies to handle test smells: a multi-method study
- What the Smell? An Empirical Investigation on the Distribution and Severity of Test Smells in Open Source Android Applications
- tsDetect: an open source test smells detection tool

1.5.2.8 Slow Component Usage

Definition:

- A component of the SUT has high latency.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- xUnit test patterns: Refactoring test code

1.5.2.9 Slow Running Tests

Definition:

- If you have unit tests that take tens of milliseconds or more to run, you have a smell.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Unit Testing Smells: What Are Your Tests Telling You?

1.5.2.10 Slow Test

Definition:

- Slow tests are kind of tests which take long enough to run

Also Known As:

- Long Running Test, The Slow Poke

Code Example:

```
class SlowTest < Test::Unit::TestCase
  def test_fast
    assert true
  end

  def test_slow
    MyClass.slow_method
    assert true
  end
end
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- A testing anti-pattern safari
- Automatic generation of smell-free unit tests

- Categorising Test Smells
- Inspecting Automated Test Code: A Preliminary Study
- Investigating into Automated Test Patterns in Erratic Tests by Considering Complex Objects
- What We Know About Smells in Software Test Code
- xUnit test patterns: Refactoring test code

1.5.2.11 The Bandwidth Demander

Definition:

- a test that cannot be successfully run when the system's running a little slowly, perhaps it has internal timing issues that are set to thresholds that are too demanding of the build process

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.5.2.12 The Slow Poke

Definition:

- Usually, it puts the test suite to execution and takes longer to finish and give the developer feedback.

Also Known As:

- Slow Test

Code Example:

```
test('shoutd show Buggy on user interaction by keyboard', done => {
    const wrapper = mount(
    <Guide
        guideContent={content}
        currentHint={0}
        showNext={false}
        invalidCode={false}
        afkExpirationTime={ 400}
        />
    );

setTimeout(() => {
    wrapper .update();
    expect (wrapper.find('BuggySleepy'). length).toBe(1);
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-Patterns - Digital Tapestry
- TDD anti patterns - Chapter 1
- TDD anti-patterns - the liar, excessive setup, the giant, slow poke
- Test-Driven Development: TDD Anti-Patterns
- Unit Testing Anti-Patterns, Full List
- Unit testing Anti-patterns catalogue

1.5.2.13 The Temporal Tip Toe

Definition:

- a test that's using timing to coordinate concurrent events, and only succeeding by good luck – ideally we should use locks and observable events to coordinate any concurrent tests, rather than hope to get our timings just right

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.5.2.14 Too Many Tests

Definition:

- There are so many tests that they are bound to take a long time to run regardless of how fast they execute.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency

- - Refactoring

- xUnit test patterns: Refactoring test code

1.5.2.15 Wait And See

Definition:

- A test that runs some set up code and then needs to 'wait' a specific amount of time before it can 'see' if the code under test functioned as expected. A testMethod that uses Thread.sleep() or equivalent is most certainly a "Wait and See" test.

Also Known As:

- The Local Hero

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-Patterns - Digital Tapestry
- Unit testing Anti-patterns catalogue

1.6 Test semantic - logic

1.6.1 Other test logic related

1.6.1.1 Anal Probe

Definition:

- A test which has to use insane, illegal or otherwise unhealthy ways to perform its task like: Reading private fields using Java's setAccessible(true) or extending a class to access protected fields/methods or having to put the test in a certain package to access package global fields/methods.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Categorising Test Smells
- Unit Testing Anti-Patterns, Full List
- Unit testing Anti-patterns catalogue

1.6.1.2 Assert 1 = 2

Definition:

- A case of developer simply yanking on the emergency brake whenever the code has dropped into a state that they didn't know how to handle.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Abap assertion anti-patterns

1.6.1.3 Asynchronous Code

Definition:

- A class cannot be tested via direct method calls. The test must start up an executable (such as a thread, process or application) and wait until it has finished starting up before interacting with it.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- xUnit test patterns: Refactoring test code

1.6.1.4 Branch To Assumption Anti-Pattern

Definition:

- An assumption is conditionally executed.

Code Example:

```
void Test(int i, int j) {
  if (i < 0)
  PexAssume.IsTrue(j > 0);
  ...
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Parameterized Test Patterns For Effective Testing with Pex

1.6.1.5 Chafing

Definition:

- A test in which the author attempts to eliminate as much textual duplication as possible, even if the indirection it introduces confuses future readers of the intention and behavior of the test.

Code Example:

```
# Subject under test
def pricing_for_code(code)
  first_factor = 0
  first_factor = 65 if code[0] == 'A'
  first_factor = 55 if code[0] == '7'
  first_factor = 40 if code[0] == '('

  second_factor = 0
  second_factor = 21 if code.size == 3
  second_factor = 19 if code.size == 5
  second_factor = 16 if code.size == 8

  return first_factor * second_factor
end

# Test
require_relative "../../../support/ruby/generate_code"

class Chafing < SmellTest
```

(continues on next page)

(continued from previous page)

```
  def test_code_one_is_correct
    code = GenerateCode.one()

    result = pricing_for_code(code)

    assert_code_pricing code, result
  end

  def test_code_two_is_correct
    code = GenerateCode.two()

    result = pricing_for_code(code)

    assert_code_pricing code, result
  end

  def test_code_three_is_correct
    code = GenerateCode.three()

    result = pricing_for_code(code)

    assert_code_pricing code, result
  end
end
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- A workbook repository of example test smells and what to do about them

1.6.1.6 Complex Conditional

Definition:

- A conditional expression is composed of many boolean conjunctions.

Code Example:

```
function calculateAmount(integer year) return float {
  var float amount;
  if (( (year mod 4) == 0 and not (year mod 100) == 0 ) or (year mod 400) == 0 ) {
    amount := BASE AMOUNT  366;
  } else {
    amount := BASE AMOUNT  365;
```

(continues on next page)

(continued from previous page)

```
  }
  return amount;
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- An approach to quality engineering of TTCN-3 test specifications
- Pattern-based Smell Detection in TTCN-3 Test Suites
- Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites

1.6.1.7 Conditional Assertions

Definition:

- it makes your test non-deterministic: you will never be sure which path will be verified in the next pass

Code Example:

```
if (existsInSystem(testUser)) {
  // test for existing user...
} else {
  // test for not existing user...
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-patterns in test automation
- On the Maintenance of System User Interactive Tests
- Smells in System User Interactive Tests
- Test Smells - The Coding Craftsman

1.6.1.8 Conditional Logic Test

Definition:

- Test methods that contain conditional statements

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Handling Test Smells in Python: Results from a Mixed-Method Study

1.6.1.9 Conditional Logic

Definition:

- Breaks the linear execution path of a test, making it less obvious which parts of the tests get executed. This increases a test's complexity and maintenance costs.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Rule-based Assessment of Test Quality

1.6.1.10 Conditional Test Logic

Definition:

- Test methods need to be simple and execute all statements in the production method. Conditions within the test method will alter the behavior of the test and its expected output, and would lead to situations where the test fails to detect defects in the production method since test statements were not executed as a condition was not met. Furthermore, conditional code within a test method negatively impacts the ease of comprehension by developers.

Also Known As:

- Indented Test Code, Guarded Test

Code Example:

```
@Test
public void testSpinner() {
    for (Map.Entry entry : sourcesMap.entrySet()) {

        String id = entry.getKey();
        Object resultObject = resultsMap.get(id);
        if (resultObject instanceof EventsModel) {
            EventsModel result = (EventsModel) resultObject;
            if (result.testSpinner.runTest) {
                System.out.println("Testing " + id + " (testSpinner)");
                //System.out.println(result);
                AnswerObject answer = new AnswerObject(entry.getValue(), "", new␣
↪CookieManager(), "");
                EventsScraper scraper = new EventsScraper(RuntimeEnvironment.application,
↪ answer);
                SpinnerAdapter spinnerAdapter = scraper.spinnerAdapter();
                assertEquals(spinnerAdapter.getCount(), result.testSpinner.data.size());
                for (int i = 0; i < spinnerAdapter.getCount(); i++) {
                    assertEquals(spinnerAdapter.getItem(i), result.testSpinner.data.
↪get(i));
                }
            }
        }
    }
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- A survey on test practitioners’ awareness of test smells
- An Exploratory Study on the Refactoring of Unit Test Files in Android Applications
- Anti-patterns of automated testing
- Automatic Identification of High-Impact Bug Report by Product and Test Code Quality
- Automatic generation of smell-free unit tests
- Categorising Test Smells
- Detection of test smells with basic language analysis methods and its evaluation
- Did You Remember To Test Your Tokens?
- Enhancing developers’ awareness on test suites’ quality with test smell summaries
- Handling Test Smells in Python: Results from a Mixed-Method Study
- How are test smells treated in the wild? A tale of two empirical studies

- Hunting for smells in natural language tests
- Improving Student Testing Practices through a Lightweight Checklist Intervention.
- Inspecting Automated Test Code: A Preliminary Study
- Investigating Severity Thresholds for Test Smells
- Investigating Test Smells in JavaScript Test Code
- On the Distribution of “Simple Stupid Bugs” in Unit Test Files: An Exploratory Study
- On the diffusion of test smells and their relationship with test code quality of Java projects
- On the distribution of test smells in open source Android applications: an exploratory study
- On the influence of Test Smells on Test Coverage
- On the test smells detection: an empirical study on the jnose test accuracy
- On the use of test smells for prediction of flaky tests
- PyNose: A Test Smell Detector For Python
- Pytest-Smell: a smell detection tool for Python unit tests
- Refactoring Test Smells With JUnit 5: Why Should Developers Keep Up-to-Date?
- Refactoring Test Smells: A Perspective from Open-Source Developers
- Rule-based Assessment of Test Quality
- Smart prediction for refactorings in the software test code
- Software Unit Test Smells
- TEMPY: Test Smell Detector for Python
- Test Smell Detection Tools: A Systematic Mapping Study
- Test code quality and its relation to issue handling performance
- TestAXE: Automatically Refactoring Test Smells Using JUnit 5 Features
- The secret life of test smells-an empirical study on test smell evolution and maintenance
- Toward static test flakiness prediction: a feasibility study
- Understanding Testability and Test Smells
- Understanding practitioners’ strategies to handle test smells: a multi-method study
- Unit Test Smells and Accuracy of Software Engineering Student Test Suites
- What the Smell? An Empirical Investigation on the Distribution and Severity of Test Smells in Open Source Android Applications
- tsDetect: an open source test smells detection tool
- xUnit test patterns: Refactoring test code
- xUnit test patterns: Refactoring test code

1.6.1.11 Conditional Tests

Definition:

- Tests are very complex and contain conditional logic that is phrased in natural language

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Hunting for smells in natural language tests

1.6.1.12 Conditional Verification Logic

Definition:

- This is usually caused by wanting to prevent the execution of assertions if the SUT fails to return the right objects or the use of loops to verify the contents of collections returned by the SUT.

Code Example:

```
public void testCombinationsOfInputValues() {
  Calculator sut = new Calculator();
  int expected;

  for (int i = 0; i < 10; i++){
    for(int j = 0 ; j < 10; j++){
      int acutal = sut.calculate(i , j);

      if(i==3 & j==4){
        expected = 8;
      } else {
        expected = i+j;
      }

      assertEquals(message(i, j), expected, actual);
    }
  }
}

private String message(int i, int j) {
  return "Cell (" + String.valueOf(i) + ", " + String.valueOf(j) + ")";
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- xUnit test patterns: Refactoring test code

1.6.1.13 Conditionals In Tests

Definition:

- having "ifs" in tests is wrong; it's proof that you need another test scenario

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-patterns of automated testing

1.6.1.14 Contaminated Test Subject

Definition:

- A test somehow morphs its subject into something less realistic for its own purposes and without regard for the resulting erosion in our confidence that the test ensures the subject's behavior under real-world conditions.

Code Example:

```
# Subject under test
require "time"

class SavingsBond
  def initialize(start_date, mature_value, term_in_years)
    @start_date = start_date
    @mature_value = mature_value
    @term_in_years = term_in_years
  end

  def current_value
    if mature?
      @mature_value
    else
      elapsed = today - @start_date
      term_duration = mature_date - @start_date
```

(continues on next page)

(continued from previous page)

```ruby
      (@mature_value * (elapsed / term_duration)).round
    end
  end
end

# Test

class ContaminatedTestSubject < SmellTest
  def setup
    @subject = SavingsBond.new(
      Time.new(2000, 5, 15),
      1000,
      10
    )
  end

  def test_when_mature_current_value_is_mature_value
    stub(@subject, :mature?, true)

    result = @subject.current_value

    assert_equal 1000, result
  end

  def test_when_immature_current_value_is_prorated
    stub(@subject, :mature?, false)
    stub(@subject, :today, Time.new(2005, 5, 15))

    result = @subject.current_value

    assert_equal 500, result
  end
end

# Fake production implementations to simplify example test of subject
class SavingsBond
  def mature?
    mature_date - Time.new > 0
  end

  def mature_date
    @start_date + term_in_seconds
  end

  def term_in_seconds
    @term_in_years * 365.25 * 24 * 60 * 60
  end

  def today
    Time.new
  end
end
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- A workbook repository of example test smells and what to do about them

1.6.1.15 Context-Dependent Rotten Green Assertion Test

Definition:

- Tests contain multiple conditional branches with different assertions in each branch

Code Example:

```
@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());
    . . .
  }

}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- RTj: a Java framework for detecting and refactoring rotten green test cases
- Rotten green tests in Java, Pharo and Python

1.6.1.16 Control Logic

Definition:

- A test method that controls test data flow by methods such as debug or halt

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Rule-based Assessment of Test Quality
- Test Smell Detection Tools: A Systematic Mapping Study

1.6.1.17 Easy Tests

Definition:

- This kind of test only checks simple things of the production code but it neglects the real logic of it.

Also Known As:

- The Dodger

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Categorising Test Smells

1.6.1.18 Embedding Implementation Detail In Your Features/Scenarios

Definition:

- Acceptance test scenarios are meant to convey intention over implementation. If you start seeing things like URLs in your test scenarios you're focusing on implementation.

Code Example:

Scenario: Social media links displayed on checkout page Given I am the checkout page for Australia Then I should see a link to 'http://twitter.com/beautifultea' And I should see a link to 'https://facebook.com/beautifultea'

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Five automated acceptance test anti-patterns

1.6.1.19 Evolve Or

Definition:

- Tests that are not maintained after SUT code evolution and still pass

Code Example:

```
public class TransactionTest {
  @Test
  public void shouldRecognizeTransactionsWithZeroValueAsInvalid() {
    //given
    Transaction tx = new Transaction(BigDecimal.ZERO,
    new InternalUser());
    //when
    boolean actual = tx.validate();
    //then
    assertThat(actual).isFalse();
  }

  @Test
  public void shouldRecognizeTransactionWithNegativeValueAsInvalid() {
    //given
    Transaction tx = new Transaction(BigDecimal.ONE.negate(),
    new InternalUser());
    //when
    boolean actual = tx.validate();
    //then
    assertThat(actual).isFalse();
  }
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Bad tests, good tests

1.6.1.20 Factories With Random Data Instead Of Sequences

Definition:

- When used alongside factories, random data generators may compromise the reliability of a test suite.

Code Example:

```
#Random factory

FactoryBot.define do
  factory :category do
    name { Faker::Lorem.word.capitalize }
  end
end
```

```
#Sequence factory

FactoryBot.define do
  factory :category do
    sequence(:name) { |n| "Category number #{n}" }
  end
end
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Rails Testing Antipatterns: Fixtures and Factories

1.6.1.21 Flexible Test

Definition:

- Test code verifies different functionality depending on when or where it is run.

Code Example:

```
public void testDisplayCurrentTime_whenever() {
  // fixture setup
  TimeDisplay sut = new TimeDisplay();

  // exercise SUT
```

(continues on next page)

(continued from previous page)

```
  String result = sut.getCurrentTimeAsHtmlFragment();

  // verify outcome
  Calendar time = new DefaultTimeProvider().getTime();
  StringBuffer expectedTime = new StringBuffer();
  expectedTime.append("<span class=\"tinyBoldText\">");

  if ((time.get(Calendar.HOUR_OF_DAY) == 0)
  && (time.get(Calendar.MINUTE) <= 1)) {
    expectedTime.append( "Midnight");
  } else if ((time.get(Calendar.HOUR_OF_DAY) == 12)
    && (time.get(Calendar.MINUTE) == 0)) { // noon
    expectedTime.append("Noon");
  } else {
    SimpleDateFormat fr = new SimpleDateFormat("h:mm a");
    expectedTime.append(fr.format(time.getTime()));
  }
  expectedTime.append("</span>");
  assertEquals( expectedTime, result);
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- xUnit test patterns: Refactoring test code

1.6.1.22 Fully Rotten Green Test

Definition:

- Contains an assertion which was forced to fail.

Code Example:

```
@Test
public void testListWindowsNewBucket() throws Exception {
  . . .
  BucketLeapArray leapArray = new BucketLeapArray(sampleCount,
          intervalInMs);
  . . . .
  List<WindowWrap<MetricBucket>> list = leapArray.list();
  for (WindowWrap<MetricBucket> wrap : list) {
    assertTrue(windowWraps.contains(wrap));
  }
```

(continues on next page)

(continued from previous page)

```
  . . . .
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- RTj: a Java framework for detecting and refactoring rotten green test cases
- Rotten green tests in Java, Pharo and Python

1.6.1.23 Generative

Definition:

- A test loops over a data structure of discrete inputs and expected outputs to generate test cases.

Code Example:

```
# Subject under test
def to_arabic(roman)
  roman.chars.each_with_index.map { |x, i|
    next_x = roman[i + 1]
    if x == "I"
      ["V", "X"].include?(next_x) ? -1 : 1
    elsif x == "V"
      next_x == "X" ? -5 : 5
    elsif x == "X"
      next_x == "C" ? -10 : 10
    end
  }.reduce(:+)
end

# Test
class Generative < SmellTest
  EXAMPLES = {
    "I" => 1,
    "II" => 2,
    "III" => 3,
    "IV" => 4,
    "V" => 5,
    "VI" => 6,
    "VII" => 7,
    "VIII" => 8,
    "IX" => 9,
    "X" => 10
```

(continues on next page)

(continued from previous page)

```
  }.each do |(roman, arabic)|
    define_method "test_roman_#{roman}_is_#{arabic}" do
      result = to_arabic(roman)

      assert_equal result, arabic
    end
  end
end
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- A workbook repository of example test smells and what to do about them

1.6.1.24 Get Really Clever And Use Random Numbers In Your Tests

Definition:

- considering using random numbers in tests as being a smell

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Smells in software test code: A survey of knowledge in industry and academia

1.6.1.25 Ground Zero

Definition:

- Where the lack of testing with 0 is the source of a lot of bugs.

Code Example:

```
// ensuring that there's a value for and then testing that it's off screen
// but if element.top is equals to 0, results goes wrong
if (element.top && (element.top < viewport.top)) {
```

(continues on next page)

(continued from previous page)

```
 hidePane();
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.6.1.26 Guarded Test

Definition:

- Guarded Tests include boolean branching logics like ifTrue: or ifFalse:

Also Known As:

- Conditional Test Logic

Code Example:

```
testRendering
  self shouldRun ifFalse: [ ^ true ].
  self assert: ...
  ...
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Assessing test quality - TestLint
- Rule-based Assessment of Test Quality
- Test Smell Detection Tools: A Systematic Mapping Study

1.6.1.27 Happy Path

Definition:

- A test that uses known input, which executes without exception and produces an expected output.

Code Example:

```
@Test
public void shouldProcessPacket() throws IOException, ServletException {
  //given
  given(request.getParameter(PacketApi.PACKET_PARAMETER))
  .willReturn(PACKET);
  given(request.getParameter(PacketApi.TYPE_PARAMETER))
  .willReturn(TYPE);
  //when
  servlet.doGet(request, response);
  //then
  verify(packetDataProcessor).process(PACKET, TYPE);
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-Patterns - Digital Tapestry
- Bad tests, good tests
- Categorising Test Smells
- Improving Student Testing Practices through a Lightweight Checklist Intervention.
- Smells in software test code: A survey of knowledge in industry and academia
- Unit Testing Anti-Patterns, Full List
- Unit testing Anti-patterns catalogue

1.6.1.28 I Wrote It Like This

Definition:

- testing the known implementation rather than the outcome of that implementation.

References:

Quality attributes

- - Code Example
- - Cause and Effect

- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.6.1.29 Incidental Details

Definition:

- Rambling long scenarios with lots of incidental details can ruin a good story. Often times the scenarios are written as a test, rather than documentation, which can lead to fluffy scenarios.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Cucumber anti-patterns (part one)

1.6.1.30 Inconsistent Wording

Definition:

- Domain concepts are not used in a consistent way (e. g., several names are used for the same domain concept).

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Hunting for smells in natural language tests

1.6.1.31 Indecisive

Definition:

- A test contains branching logic. Of course, test-scoped logic is always risky, since it is itself untested. But this smell portends some deeper issues worth discussing.

Code Example:

```
# Subject under test
require 'rbconfig'

def join_path(fragments)
  if /mswin/ =~ RbConfig::CONFIG['host_os']
    separator = "\\"
    pattern = /\\+/
  else
    separator = "/"
    pattern = /\/+/
  end
  fragments.join(separator).gsub(pattern, separator)
end

# Test
class Indecisive < SmellTest
  def test_simple_case
    fragments = ["foo", "bar", "baz"]

    result = join_path(fragments)

    if /mswin/ =~ RbConfig::CONFIG['host_os']
      assert_equal "foo\\bar\\baz", result
    else
      assert_equal "foo/bar/baz", result
    end
  end

  def test_contains_separators
    if /mswin/ =~ RbConfig::CONFIG['host_os']
      fragments = ["\\foo\\", "bar\\biz", "baz\\"]
    else
      fragments = ["/foo/", "bar/biz", "baz/"]
    end

    result = join_path(fragments)

    if /mswin/ =~ RbConfig::CONFIG['host_os']
      assert_equal "\\foo\\bar\\biz\\baz\\", result
    else
      assert_equal "/foo/bar/biz/baz/", result
    end
  end
end
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- A workbook repository of example test smells and what to do about them

1.6.1.32 Indented Test

Definition:

- A test method that contains a large number of decision points, loops, and conditional statements

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smell Detection Tools: A Systematic Mapping Study
- TestQ: Exploring Structural and Maintenance Characteristics of Unit Test Suites

1.6.1.33 Insufficient Grouping

Definition:

- A module or group contains too many elements. Especially for large modules, groups should be used to add logical structure to the module and enhance readability. If a group reaches a critical size, it can be structured further by subgroups.

Code Example:

```
module InsufficientGrouping {
 type record myRecordType1 { // . . .
 }

 type record myRecordType2 { // . . .
 }

 type record myRecordType3 { // . . .
 }

 type record myRecordType4 { // . . .
 }
```

(continues on next page)

(continued from previous page)

```
 type record myRecordType5 { // . . .
 }

 template myRecordType1 myTemplate1 := { // . . .
 }

 template myRecordType1 myTemplate2 := { // . . .
 }

 template myRecordType1 myTemplate3 := { // . . .
 }

 template myRecordType1 myTemplate4 := { // . . .
 }

 template myRecordType1 myTemplate5 := { // . . .
 }

 function f1( ) { // . . .
 }

// more declarations here
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- An approach to quality engineering of TTCN-3 test specifications
- Pattern-based Smell Detection in TTCN-3 Test Suites
- Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites

1.6.1.34 Invalid Test Data

Definition:

- when the test data would not be valid if used in real life – does this make the test invalid or not?

References:

Quality attributes

- - Code Example

- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.6.1.35 Invasion Of Privacy

Definition:

- A test is written directly against a method that's intended to be a private implementation detail of the subject.

Code Example:

```ruby
# Subject under test
class SeatMap
  def initialize(ticket, original_seat)
    @fare_class = ticket.fare_class
    @current_seat = original_seat
  end

  def moveTo(new_seat)
    raise "No seat selected" unless new_seat
    raise "Invalid seat selected" unless /^\d\d?[A-J]$/.test(new_seat)

    if qualify_fare_class_for_seat(new_seat)
      @current_seat = new_seat
    else
      raise Error "Seat not available for ticket's fare class"
    end
  end

  private

  # Private API! Don't call this!
  def qualify_fare_class_for_seat(seat)
    return unless seat

    if row_match = seat.match(/^(\d+)/)
      row = row_match[0].to_i
      row > 10
    end
  end
end

# Test
class InvasionOfPrivacy < SmellTest
  def setup
    @ticket = OpenStruct.new(fare_class: "M")
    @subject = SeatMap.new(@ticket, "18D")
    super
  end
```

(continues on next page)

(continued from previous page)

```
  def test_ensure_seat_not_null
    result = @subject.send(:qualify_fare_class_for_seat, nil)

    refute result
  end

  def test_do_not_break_if_seat_lacks_a_row_number
    result = @subject.send(:qualify_fare_class_for_seat, "A")

    refute result
  end

  def test_approve_if_behind_row_ten
    result = @subject.send(:qualify_fare_class_for_seat, "11B")

    assert_equal true, result
  end

  def test_deny_if_ahead_of_row_ten
    result = @subject.send(:qualify_fare_class_for_seat, "9J")

    refute result
  end
end
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- A workbook repository of example test smells and what to do about them
- Smells in software test code: A survey of knowledge in industry and academia

1.6.1.36 Lazy Test

Definition:

- More than one test case with the same fixture that tests the same method. This smell affects test maintainability, as assertions testing the same method should be in the same test case. Like EAGER TEST, the original definition [3] leaves some details to interpretation. We consider every call to the class under test as a potential cause of LAZY TEST, irrespective of whether their results are used in an assertion.

Code Example:

```
@Test
public void testDecrypt() throws Exception {
    FileInputStream file = new FileInputStream(ENCRYPTED_DATA_FILE_4_14);
    byte[] enfileData = new byte[file.available()];
    FileInputStream input = new FileInputStream(DECRYPTED_DATA_FILE_4_14);
    byte[] fileData = new byte[input.available()];
    input.read(fileData);
    input.close();
    file.read(enfileData);
    file.close();
    String expectedResult = new String(fileData, "UTF-8");
    assertEquals("Testing simple decrypt",expectedResult, Cryptographer.
↪decrypt(enfileData, "test"));
}

@Test
public void testEncrypt() throws Exception {
    String xml = readFileAsString(DECRYPTED_DATA_FILE_4_14);
    byte[] encrypted = Cryptographer.encrypt(xml, "test");
    String decrypt = Cryptographer.decrypt(encrypted, "test");
    assertEquals(xml, decrypt);
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- An Empirical Study into the Relationship Between Class Features and Test Smells
- An Exploratory Study on the Refactoring of Unit Test Files in Android Applications
- An empirical analysis of the distribution of unit test smells and their impact on software maintenance
- An exploratory study of the relationship between software test smells and fault-proneness
- Are test smells really harmful? An empirical study
- Assessing diffusion and perception of test smells in scala projects
- Automatic Identification of High-Impact Bug Report by Product and Test Code Quality
- Automatic generation of smell-free unit tests
- Categorising Test Smells
- Developers perception on the severity of test smells: an empirical study
- Enhancing developers’ awareness on test suites’ quality with test smell summaries
- Handling Test Smells in Python: Results from a Mixed-Method Study
- Investigating Test Smells in JavaScript Test Code

- On the diffusion of test smells and their relationship with test code quality of Java projects
- On the distribution of test smells in open source Android applications: an exploratory study
- On the influence of Test Smells on Test Coverage
- On the interplay between software testing and evolution and its effect on program comprehension
- On the test smells detection: an empirical study on the jnose test accuracy
- On the use of test smells for prediction of flaky tests
- Refactoring Test Code
- Refactoring Test Smells With JUnit 5: Why Should Developers Keep Up-to-Date?
- SoCRATES: Scala radar for test smells
- Software Unit Test Smells
- Test Smell Detection Tools: A Systematic Mapping Study
- The secret life of test smells-an empirical study on test smell evolution and maintenance
- What the Smell? An Empirical Investigation on the Distribution and Severity of Test Smells in Open Source Android Applications
- tsDetect: an open source test smells detection tool

1.6.1.37 London School Orthodoxy

Definition:

- Unit tests should primarily target public APIs. Only this way can you optimize the encapsulated innards without constantly having to update the tests. Test suites that get in the way of refactoring are often tightly coupled to implementation details.

Code Example:

- No code examples yet…

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Testing the Untestable and Other Anti-Patterns

1.6.1.38 Multiple Test Conditions

Definition:

- A test is trying to apply the same test logic to many sets of input values each with their own corresponding expected result.

Code Example:

```
public void testMultipleValueSets(){
  // Set Up Fixture
  Calculator sut = new Calculator();
  TestValues[] testValues = {
    new TestValues(1,2,3),
    new TestValues(2,3,5),
    new TestValues(3,4,8),
    new TestValues(4,5,9)
  };
  for (int i = 0; i < testValues.length; i++){
    TestValues values = testValues[i];
    int actual = sut.calculate( values.a, values.b);
    assertEquals(message(i), values.expectedSum, actual);
  }
}

private String message(int i) {
  return "Row "+ String.valueOf(i);
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- xUnit test patterns: Refactoring test code

1.6.1.39 Nested Conditional

Definition:

- Nested conditional expression. Use if and else leg of a conditional only if both paths are part of the normal behavior; if one leg is an unusual condition, use a separate exit point (guard clause) instead.

Code Example:

References:

Quality attributes

- - Code Example

- - Cause and Effect
- - Frequency
- - Refactoring

- An approach to quality engineering of TTCN-3 test specifications
- Pattern-based Smell Detection in TTCN-3 Test Suites
- Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites

1.6.1.40 Neverfail Test

Definition:

- We may just "know" that some piece of functionality is not working but the tests for that functionality are passing nonetheless. When doing test-driven development we have added a test for functionality we have not yet written but we cannot get it to fail.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- xUnit test patterns: Refactoring test code

1.6.1.41 Only Easy Tests

Definition:

- Similar to happy path tests, easy tests concentrate on things that are easy to verify, such as simple property values. However, the real logic of the unit under test is ignored. This is a symptom of an inexperienced developer trying to test complex code.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- JUnit Anti-patterns

1.6.1.42 Only Happy Path Tests

Definition:

- Only the expected behaviour of the system is tested. Valid input data is fed into the system, and the results are checked against the expected correct answer. What is missing in this case is the exceptional conditions.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- JUnit Anti-patterns

1.6.1.43 Paranoid

Definition:

- A test (and as a result, its subject) covers edge cases that aren't actually reachable by the production application.

Code Example:

```ruby
# Subject under test
require "json"

# [Note: In production, this func will only ever be called by .csv_for below]
def escape_csv_string_value(value)
  if !value
    ""
  elsif !value.kind_of?(String)
    escape_csv_string_value(JSON.dump(value))
  elsif value.include?(",")
    "\"#{value.gsub(/"/, "\"\"")}\""
  else
    value
  end
end

# Test
class Paranoid < SmellTest
  def test_does_nothing_without_commas
    text = "hi my name is \"Todd\""

    result = escape_csv_string_value(text)

    assert_equal text, result
  end

  def test_wraps_comma_strings_with_double_quotes
```

(continues on next page)

(continued from previous page)

```
    result = escape_csv_string_value("Hello, world.")

    assert_equal "\"Hello, world.\"", result
  end

  def test_double_quotes_in_comma_strings
    result = escape_csv_string_value("Hello, \"Todd\".")

    assert_equal "\"Hello, \"\"Todd\"\".\"", result
  end

  def test_ensure_not_null
    result = escape_csv_string_value(nil)

    assert_equal "", result
  end

  def test_coerce_number_to_string
    result = escape_csv_string_value(42)

    assert_equal "42", result
  end

  def test_coerce_object_to_string_and_escape_the_string
    result = escape_csv_string_value(name: "Jim", age: 64)

    assert_equal "\"{\"\"name\"\":\"\"Jim\"\",\"\"age\"\":64}\"", result
  end
end

# Fake production implementation to add context to the subject
def csv_for(value)
  if value.kind_of?(String)
    escape_csv_string_value(value)
  elsif value.kind_of?(Array)
    if value.any? { |item| item.kind_of?(Array) }
      value.map { |item| csv_for(item) }.join("\n")
    else
      value.map { |item| csv_for(item) }.join(",")
    end
  else
    value
  end
end

# Example use:
csv_for([["id", "name", "bio"], [1, "joe", "why, hello \"joe\"!"]])
# => 'id,name,bio\n1,joe,"why, hello ""joe""!"'
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- A workbook repository of example test smells and what to do about them

1.6.1.44 Parsed Data

Definition:

- A test creates structured data by parsing unstructured input and only uses the structured data during the test.

Code Example:

```
void ParseAndTest(string xml) {
  // parse
  Employee e = Employee.Deserialize(xml);
  // test logic
  EmployeeCollection c = new EmployeeCollection();
  c.Add(e);
  Assert.IsTrue(c.Contains(e));
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Parameterized Test Patterns For Effective Testing with Pex

1.6.1.45 Quixotic

Definition:

- A test that charts an idealistic path through the subject code, cherry-picking inputs that provide minimum resistance (e.g. in test data setup), which may result in missed test coverage in code that handle negative cases. Notably, this is more likely to occur when those negative cases are also somehow complex, which is precisely when good testing is important!

Code Example:

```
# Subject under test
def rank_hotel_review(user, title, text, stars)
  rank = 10
  rank -= 3 unless user.logged_in
```

(continues on next page)

(continued from previous page)

```ruby
  rank += 10 if stars == 5 || stars == 1
  if rank > 1 && obscene?(title)
    raise "Underage swearing!" unless user.age > 13
    rank = 1
  elsif rank > 3 && obscene?(text)
    rank = 3 unless user.occupation == "sailor"
  end
  return rank
end

# Test
class Quixotic < SmellTest
  def test_5_star_member
    user = OpenStruct.new(logged_in: true)

    result = rank_hotel_review(user, "title", "body", 5)

    assert_equal 20, result
  end

  def test_3_star_anonymous
    user = OpenStruct.new(logged_in: false)

    result = rank_hotel_review(user, "title", "body", 3)

    assert_equal 7, result
  end
end

# Fake production implementations to simplify example test of subject
def obscene?(text)
  text.include?("obscenities")
end
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- A workbook repository of example test smells and what to do about them

1.6.1.46 Rewriting Private Methods As Public

Definition:

- This anti-pattern arises when a developer is trying to increase code coverage but is not able to test all of the private methods in a class

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-Patterns In Unit Testing

1.6.1.47 Rotten Green Test

Definition:

- A test that passes (is green) but contains assertions that are never executed

Code Example:

```
@Test
public void testListWindowsNewBucket() throws Exception {
  . . .
  BucketLeapArray leapArray = new BucketLeapArray(sampleCount,
          intervalInMs);
  . . . .
  List<WindowWrap<MetricBucket>> list = leapArray.list();
  for (WindowWrap<MetricBucket> wrap : list) {
    assertTrue(windowWraps.contains(wrap));
  }
  . . . .
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Automatic generation of smell-free unit tests
- Detection of test smells with basic language analysis methods and its evaluation

- JTDog: a Gradle Plugin for Dynamic Test Smell Detection
- RTj: a Java framework for detecting and refactoring rotten green test cases
- Test Smell Detection Tools: A Systematic Mapping Study

1.6.1.48 Skip Rotten Green Test

Definition:

- Contains guards to stop their execution early under certain conditions.

Code Example:

```
@Test public void testNormalizedKeyReadWriter(){
   . . .
   TypeComparator <T> comp1 = getComparator(true);

   if(!comp1.supportsSerializationWithKeyNormalization()){
    return ;
   }
   . . .
   assertTrue(comp1.compareToReference(comp2) == 0);
   . . .
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- RTj: a Java framework for detecting and refactoring rotten green test cases

1.6.1.49 Skip-Epidemic

Definition:

- Failing tests are constantly skipped over.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Categorising Test Smells

1.6.1.50 Sneaky Checking

Definition:

- The test hides its assertions in actions that are at the wrong level of details.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- On the Maintenance of System User Interactive Tests
- Smells in System User Interactive Tests
- Test design for automation: Anti-patterns

1.6.1.51 Success Against All Odds

Definition:

- A test that was written to pass first rather than fail first. As an unfortunate side effect, the test case happens to always pass even though the test should fail.

Also Known As:

- The Liar

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-Patterns - Digital Tapestry
- Categorising Test Smells
- TDD and anti-patterns - Chapter 5
- Test-Driven Development: TDD Anti-Patterns

1.6.1.52 Tangential

Definition:

- The subject and its test claim to be focused on something, but the bulk of their complexity is focused on a different (often subordinate) responsibility.

Code Example:

```
class Tangential < SmellTest
  def test_no_type
    user = OpenStruct.new

    set_attr(user, :name, "Fred")

    assert_equal "Fred", user.name
  end

  def test_string_type_correct
    user = OpenStruct.new

    set_attr(user, :name, "Frida", :string)

    assert_equal "Frida", user.name
  end

  def test_string_type_incorrect
    user = OpenStruct.new

    error = assert_raises {
      set_attr(user, :age, 42, :string)
    }
    assert_equal "42 is not a string", error.message
  end

  def test_phone_type_correct
    user = OpenStruct.new

    set_attr(user, :mobile, "(614) 349-4279", :phone)

    assert_equal "(614) 349-4279", user.mobile
  end

  def test_phone_type_incorrect
    user = OpenStruct.new

    error = assert_raises {
      set_attr(user, :mobile, "1337", :phone)
    }
    assert_equal "1337 is not a phone", error.message
  end

  def test_invalid_first_phone_character_cannot_start_with_1
    user = OpenStruct.new
```

(continues on next page)

(continued from previous page)

```
    assert_raises {
      set_attr(user, :mobile, "(123) 456-7890", :phone)
    }
  end

  def test_simple_japanese_phone_number
    user = OpenStruct.new

    set_attr(user, :mobile, "090-1790-1357", :phone)

    assert_equal "090-1790-1357", user.mobile
  end

  def test_japanese_without_the_trunk
    user = OpenStruct.new

    assert_raises {
      set_attr(user, :mobile, "90-1790-1357", :phone)
    }
  end

  def test_international_japanese_phone_number
    user = OpenStruct.new

    set_attr(user, :mobile, "011-81-90-1790-1357", :phone)

    assert_equal "011-81-90-1790-1357", user.mobile
  end
end
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- A workbook repository of example test smells and what to do about them

1.6.1.53 Test By Number

Definition:

- Production code is tested consistently by rote, inadvertently suppressing creativity in the design of both the tests and their subjects.

Code Example:

```
# Test
class TestByNumber < SmellTest
  def setup
    @repo = Repo.new
    @subject = AddressController.new(@repo)
    super
  end

  def test_create
    user = @repo.save(OpenStruct.new(name: 'Jane', addresses: []))
    address_params = OpenStruct.new(street: 'some street')

    @subject.create(user, address_params)

    address = @repo.find(address_params.id)
    assert_equal "some street", address.street
    assert_equal address, user.addresses[0]
  end

  def test_read
    address = @repo.save(OpenStruct.new(street: 'a street'))
    user = @repo.save(OpenStruct.new(name: 'Joe', addresses: [address]))

    result = @subject.read(user, address.id)

    assert_equal address, result
  end

  def test_update
    address = @repo.save(OpenStruct.new(street: 'no street', zip: '12345'))
    user = @repo.save(OpenStruct.new(name: 'Jill', addresses: [address]))

    @subject.update(user, {
      id: address.id,
      street: 'the street'
    })

    address = @repo.find(address.id)
    assert_equal "the street", address.street
    assert_equal "12345", address.zip
  end

  def test_destroy
    address = @repo.save(OpenStruct.new(street: 'foo street'))
    user = @repo.save(OpenStruct.new(name: 'Gene', addresses: [address]))
```

(continues on next page)

(continued from previous page)

```
    @subject.destroy(user, address.id)

    assert_equal nil, @repo.find(address.id)
    assert_equal 0, user.addresses.length
  end

  # Fake production implementations to simplify example test of subject
  class Repo
    def initialize
      @items = {}
      @next_id = 1
    end

    def save(item)
      item.id = item.id || @next_id += 1
      @items[item.id] = item
      return item
    end

    def find(item_id)
      @items[item_id]
    end

    def destroy(item_id)
      @items.delete(item_id)
    end
  end
end
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- A workbook repository of example test smells and what to do about them

1.6.1.54 Test-Per-Method

Definition:

- Although a one-to-one relationship between test and production classes is a reasonable starting point, a one-to-one relationship between test and production method is almost always a bad idea

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Unit Testing Anti-Patterns, Full List

1.6.1.55 Testing Internal Implementation

Definition:

- Related to tests that are tightly coupled to internal implementation

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Software Testing Anti-patterns

1.6.1.56 Testing The Authentication Framework

Definition:

- When the test literally compares the generated message digest with a string constant to see if both are identical and they are provided by a 3rd party library, whose maintainers are responsible to properly test the code and not the developer.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency

- - Refactoring

- Did You Remember To Test Your Tokens?

1.6.1.57 Tests That Can'T Fail

Definition:

- TDD tests that pass rather than failing in the first execution.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- A testing anti-pattern safari

1.6.1.58 The Dodger

Definition:

- A unit test which has lots of tests for minor (and presumably easy to test) side effects, but never tests the core desired behavior. Sometimes you may find this in database access related tests, where a method is called, then the test selects from the database and runs assertions against the result.

Also Known As:

- Easy Tests

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-Patterns - Digital Tapestry
- Test-Driven Development: TDD Anti-Patterns
- Unit Testing Anti-Patterns, Full List

1.6.1.59 The Inspector

Definition:

- A unit test that violates encapsulation in an effort to achieve 100% code coverage, but knows so much about what is going on in the object that any attempt to refactor will break the existing test and require any change to be reflected in the unit test.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-Patterns - Digital Tapestry
- Categorising Test Smells
- Test-Driven Development: TDD Anti-Patterns
- Unit Testing Anti-Patterns, Full List
- Unit testing Anti-patterns catalogue

1.6.1.60 The Liar

Definition:

- Testing asynchronous code becomes tricky as it is based on a future value that you might receive or might not.

Also Known As:

- Evergreen Tests, Success Against All Odds

Code Example:

```
test('the data is peanut butter', () => {
  function callback(data) {
    expect(data).toBe('peanut butter');
  }

  fetchData(callback);
});
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-Patterns - Digital Tapestry
- Categorising Test Smells
- Smells in software test code: A survey of knowledge in industry and academia
- TDD anti patterns - Chapter 1
- TDD anti-patterns - the liar, excessive setup, the giant, slow poke
- Test-Driven Development: TDD Anti-Patterns
- Unit Testing Anti-Patterns, Full List

1.6.1.61 The Sequencer

Definition:

- A unit test that depends on items in an unordered list appearing in the same order during assertions.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-Patterns - Digital Tapestry
- Categorising Test Smells
- Test-Driven Development: TDD Anti-Patterns
- Unit Testing Anti-Patterns, Full List

1.6.1.62 Underspecification

Definition:

- While it's clearly possible for a test suite to do too much, it's far more common for it to do too little.

Code Example:

```
MIN_FREE_SHIPPING_PRICE = 25.0

def free_shipping?(total_order_price)
total_order_price > MIN_FREE_SHIPPING_PRICE
end
```

```
def test_free_shipping_returns_true_for_order_above_min_price
assert free_shipping?(MIN_FREE_SHIPPING_PRICE + 1)
end
```

(continues on next page)

(continued from previous page)

```
def test_free_shipping_returns_false_for_order_below_min_price
assert !free_shipping?(MIN_FREE_SHIPPING_PRICE - 1)
end
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Testing anti-patterns: How to fail with 100% test coverage

1.6.1.63 Untestable Test Code

Definition:

- The body of a Test Method is obscure enough or contains enough Conditional Test Logic to cause us to wonder whether the test is correct.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- xUnit test patterns: Refactoring test code

1.6.1.64 Use Smart Values

Definition:

- Tests pass when the scenario they test is not really fulfilled

Code Example:

```
public class PriceCalculatorFactoryTest {
  SettingsService settings = mock(SettingsService.class);
  @Test
  public void shouldCreatePriceCalculator() {
    //given
    given(settings.getMinMargin()).willReturn(new BigDecimal(20));
    given(settings.getMaxMargin()).willReturn(new BigDecimal(50));
    given(settings.getPremiumShare()).willReturn(new BigDecimal(50));
```

(continues on next page)

(continued from previous page)

```
    //when
    PriceCalculator calculator
    = new PriceCalculatorFactory(settings).create();
    Chapter 3. Strength
    7
    //then
    assertThat(calculator)
    .isEqualTo(new PriceCalculator(new BigDecimal(20),
    new BigDecimal(50), new BigDecimal(50)));
  }
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Bad tests, good tests

1.6.1.65 Wheel Of Fortune

Definition:

- where random values in the test can lead to error – see also It Passed Yesterday

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.6.1.66 X-Ray Specs

Definition:

- A test that accesses or edits private, internal state of the subject that it shouldn't logically be privy to.

Code Example:

```ruby
# Subject under test
class MutableSeatMap
  attr_reader :approvals, :current_seat

  def initialize(ticket, original_seat)
    @fare_class = ticket.fare_class
    @current_seat = original_seat
    @approvals = {}
  end

  def move_to(new_seat)
    if !@approvals[@fare_class] || !@approvals[@fare_class][new_seat]
      qualify_fare_class_for_seat!(new_seat)
    end

    if @approvals[@fare_class][new_seat]
      @current_seat = new_seat
    end
  end
end

# Test
class XRaySpecs < SmellTest
  def setup
    @ticket = OpenStruct.new(fare_class: "M")
    @subject = MutableSeatMap.new(@ticket, "18D")
    super
  end

  def test_approve_if_behind_row_ten
    @subject.move_to("11B")

    assert_equal true, @subject.approvals["M"]["11B"]
    assert_equal "11B", @subject.current_seat, "11B"
  end

  def test_deny_if_ahead_of_row_ten
    @subject.move_to("9J")

    assert_equal false, @subject.approvals["M"]["9J"]
    assert_equal "18D", @subject.current_seat
  end

  def test_will_short_circuit_approval_process_when_memoized
    @subject.approvals["M"] = {"Havanna" => "Sure, why not"}
```

(continues on next page)

(continued from previous page)

```
    @subject.move_to("Havanna")

    assert_equal "Sure, why not", @subject.approvals["M"]["Havanna"]
    assert_equal "Havanna", @subject.current_seat
  end
end

# Fake production implementations to simplify example test of subject
class MutableSeatMap
  def qualify_fare_class_for_seat!(seat)
    allowed = seat.match(/^(\d+)/)[0].to_i > 10
    @approvals[@fare_class] ||= {}
    @approvals[@fare_class][seat] = allowed
  end
end
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- A workbook repository of example test smells and what to do about them
- Smells in software test code: A survey of knowledge in industry and academia

1.6.1.67 You Do Weird Things To Get At The Code Under Test

Definition:

- The use of reflection schemes to invoke and test private methods.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Unit Testing Smells: What Are Your Tests Telling You?

1.6.2 Testing many things

1.6.2.1 Assert The World

Definition:

- where the assertions prove everything, even uninteresting stuff.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Test Smells - The Coding Craftsman

1.6.2.2 Assertion Roulette

Definition:

- Occurs when a test method has multiple non-documented assertions. Multiple assertion statements in a test method without a descriptive message impacts readability/understandability/maintainability as it's not possible to understand the reason for the failure of the test.

```python
import unittest

airLinesCode = ['2569','2450','2340']

class Flight:
    def __init__(self,airLine,mileage):
        self.mileage = mileage
        self.airLine = airLine
        self.fullFuel = True

    def isValidAirLineCode(self):
        for airLineCode in airLinesCode:
            if(self.airLine == airLineCode):
                return True
        return False

class TestFlight(unittest.TestCase):
    def test_flight(self):
        flight = Flight('2569',1000)

        self.assertEqual(flight.mileage,1000)
        self.assertTrue(flight.fullFuel)
        self.assertTrue(flight.isValidAirLineCode())

```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    unittest.main(argv=['first-arg-is-ignored'], exit=False)
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- A preliminary evaluation on the relationship among architectural and test smells
- A survey on test practitioners' awareness of test smells
- An Empirical Study into the Relationship Between Class Features and Test Smells
- An Exploratory Study on the Refactoring of Unit Test Files in Android Applications
- An empirical analysis of the distribution of unit test smells and their impact on software maintenance
- An empirical investigation into the nature of test smells
- An exploratory study of the relationship between software test smells and fault-proneness
- Analyzing Test Smells Refactoring from a Developers Perspective
- Are test smells really harmful? An empirical study
- As Code Testing: Characterizing Test Quality in Open Source Ansible Development
- Assessing diffusion and perception of test smells in scala projects
- Automatic Identification of High-Impact Bug Report by Product and Test Code Quality
- Automatic generation of smell-free unit tests
- Categorising Test Smells
- Characterizing High-Quality Test Methods: A First Empirical Study
- Detection of test smells with basic language analysis methods and its evaluation
- Developers perception on the severity of test smells: an empirical study
- Did You Remember To Test Your Tokens?
- Do the Test Smells Assertion Roulette and Eager Test Impact Students' Troubleshooting and Debugging Capabilities?
- Enhancing developers' awareness on test suites' quality with test smell summaries
- Generated Tests in the Context of Maintenance Tasks: A Series of Empirical Studies
- Handling Test Smells in Python: Results from a Mixed-Method Study
- How are test smells treated in the wild? A tale of two empirical studies
- Improving Student Testing Practices through a Lightweight Checklist Intervention.
- Inspecting Automated Test Code: A Preliminary Study

- Investigating Severity Thresholds for Test Smells
- Investigating Test Smells in JavaScript Test Code
- Is Assertion Roulette still a test smell? An experiment from the perspective of testing education
- On the Distribution of "Simple Stupid Bugs" in Unit Test Files: An Exploratory Study
- On the Relation of Test Smells to Software Code Quality
- On the diffusion of test smells and their relationship with test code quality of Java projects
- On the diffusion of test smells in automatically generated test code: an empirical study
- On the distribution of test smells in open source Android applications: an exploratory study
- On the influence of Test Smells on Test Coverage
- On the interplay between software testing and evolution and its effect on program comprehension
- On the test smells detection: an empirical study on the jnose test accuracy
- On the use of test smells for prediction of flaky tests
- Pizza versus Pinsa: On the Perception and Measurability of Unit Test Code Quality
- Practitioner Perceptions of Ansible Test Smells
- PyNose: A Test Smell Detector For Python
- Pytest-Smell: a smell detection tool for Python unit tests
- Quality defects detection in unit tests
- RAIDE: a tool for Assertion Roulette and Duplicate Assert identification and refactoring
- Refactoring Test Code
- Refactoring Test Smells With JUnit 5: Why Should Developers Keep Up-to-Date?
- Refactoring Test Smells: A Perspective from Open-Source Developers
- Revisiting Test Smells in Automatically Generated Tests: Limitations, Pitfalls, and Opportunities
- Scented since the beginning: On the diffuseness of test smells in automatically generated test code
- Smart prediction for refactorings in the software test code
- SoCRATES: Scala radar for test smells
- Software Unit Test Smells
- Test Artifacts — The Practical Testing Book
- Test Smell Detection Tools: A Systematic Mapping Study
- Test-related factors and post-release defects: an empirical study
- TestAXE: Automatically Refactoring Test Smells Using JUnit 5 Features
- TestQ: Exploring Structural and Maintenance Characteristics of Unit Test Suites
- The Relation of Test-Related Factors to Software Quality: A Case Study on Apache Systems
- The secret life of test smells-an empirical study on test smell evolution and maintenance
- To What Extent Can Code Quality be Improved by Eliminating Test Smells?
- Toward static test flakiness prediction: a feasibility study
- Towards Automated Tools for Detecting Test Smells: An Empirical Investigation into the Nature of Test Smells

- Understanding Testability and Test Smells
- Understanding practitioners' strategies to handle test smells: a multi-method study
- Unit Test Smells and Accuracy of Software Engineering Student Test Suites
- What We Know About Smells in Software Test Code
- What the Smell? An Empirical Investigation on the Distribution and Severity of Test Smells in Open Source Android Applications
- Why do builds fail?—A conceptual replication study
- tsDetect: an open source test smells detection tool
- xUnit test patterns: Refactoring test code

1.6.2.3 Eager Test

Definition:

- A test case that checks or uses more than one method of the class under test. Since its introduction, this smell has been somewhat broadly defined. It is left to interpretation which method calls count towards the maximum. Either all methods invoked on the class under test could count, or only the methods invoked on the same instance under test, or only the methods of which the return value is eventually used within an assertion.

Also Known As:

- The Test It All, Split Personality, Many Assertions, Multiple Assertions, The Free Ride, The One, Piggyback, Silver Bullet

Code Example:

```java
public void testFlightMileage_asKm2() throws Exception {
     // setup fixture
     // exercise contructor
     Flight newFlight = new Flight(validFlightNumber);
     // verify constructed object
     assertEquals(validFlightNumber, newFlight.number);
     assertEquals("", newFlight.airlineCode);
     assertNull(newFlight.airline);
     // setup mileage
     newFlight.setMileage(1122);
     // exercise mileage translater
     int actualKilometres = newFlight.getMileageAsKm();
     // verify results
     int expectedKilometres = 1810;
     assertEquals( expectedKilometres, actualKilometres);
     // now try it with a canceled flight:
     newFlight.cancel();
     try {
         newFlight.getMileageAsKm();
         fail("Expected exception");
     } catch (InvalidRequestException e) {
         assertEquals( "Cannot get cancelled flight mileage", e.getMessage());
     }
 }
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- A preliminary evaluation on the relationship among architectural and test smells
- A survey on test practitioners' awareness of test smells
- An Empirical Study into the Relationship Between Class Features and Test Smells
- An Exploratory Study on the Refactoring of Unit Test Files in Android Applications
- An analysis of information needs to detect test smells
- An empirical analysis of the distribution of unit test smells and their impact on software maintenance
- An empirical investigation into the nature of test smells
- An exploratory study of the relationship between software test smells and fault-proneness
- Analyzing Test Smells Refactoring from a Developers Perspective
- Are test smells really harmful? An empirical study
- Assessing diffusion and perception of test smells in scala projects
- Automatic Identification of High-Impact Bug Report by Product and Test Code Quality
- Automatic Refactoring Method to Remove Eager Test Smell
- Automatic Test Smell Detection Using Information Retrieval Techniques
- Automatic generation of smell-free unit tests
- Categorising Test Smells
- Characterizing the Relative Significance of a Test Smell
- Detecting redundant unit tests for AspectJ programs
- Developers perception on the severity of test smells: an empirical study
- Do the Test Smells Assertion Roulette and Eager Test Impact Students' Troubleshooting and Debugging Capabilities?
- Enhancing developers' awareness on test suites' quality with test smell summaries
- Generated Tests in the Context of Maintenance Tasks: A Series of Empirical Studies
- Handling Test Smells in Python: Results from a Mixed-Method Study
- How are test smells treated in the wild? A tale of two empirical studies
- Investigating Severity Thresholds for Test Smells
- Investigating Test Smells in JavaScript Test Code
- Just-In-Time Test Smell Detection and Refactoring: The DARTS Project
- Machine Learning-Based Test Smell Detection

- Obscure Test
- On The Detection of Test Smells: A Metrics-Based Approach for General Fixture and Eager Test
- On the Relation of Test Smells to Software Code Quality
- On the diffusion of test smells and their relationship with test code quality of Java projects
- On the diffusion of test smells in automatically generated test code: an empirical study
- On the distribution of test smells in open source Android applications: an exploratory study
- On the influence of Test Smells on Test Coverage
- On the interplay between software testing and evolution and its effect on program comprehension
- On the test smells detection: an empirical study on the jnose test accuracy
- On the use of test smells for prediction of flaky tests
- Pizza versus Pinsa: On the Perception and Measurability of Unit Test Code Quality
- Pytest-Smell: a smell detection tool for Python unit tests
- Refactoring Test Code
- Revisiting Test Smells in Automatically Generated Tests: Limitations, Pitfalls, and Opportunities
- Rule-based Assessment of Test Quality
- Scented since the beginning: On the diffuseness of test smells in automatically generated test code
- Smart prediction for refactorings in the software test code
- Smells in software test code: A survey of knowledge in industry and academia
- SoCRATES: Scala radar for test smells
- Software Unit Test Smells
- Test Smell Detection Tools: A Systematic Mapping Study
- Test code quality and its relation to issue handling performance
- Test-related factors and post-release defects: an empirical study
- TestQ: Exploring Structural and Maintenance Characteristics of Unit Test Suites
- The Relation of Test-Related Factors to Software Quality: A Case Study on Apache Systems
- The secret life of test smells-an empirical study on test smell evolution and maintenance
- To What Extent Can Code Quality be Improved by Eliminating Test Smells?
- Toward static test flakiness prediction: a feasibility study
- Towards Automated Tools for Detecting Test Smells: An Empirical Investigation into the Nature of Test Smells
- Understanding Testability and Test Smells
- Unit Test Smells and Accuracy of Software Engineering Student Test Suites
- What We Know About Smells in Software Test Code
- What the Smell? An Empirical Investigation on the Distribution and Severity of Test Smells in Open Source Android Applications
- tsDetect: an open source test smells detection tool
- xUnit test patterns: Refactoring test code

1.6.2.4 Field-Level Assertion

Definition:

- This occurs when the individual fields written to the database or in output arrays have their own assertions.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Design Patterns for Database API Testing 1: Web Service Saving 2 – Code

1.6.2.5 Many Assertions

Definition:

- When a test tests way too much its failure is often hard to track down.

Also Known As:

- Eager Test

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Smells of Testing (signs your tests are bad)

1.6.2.6 Method-Based Testing

Definition:

- This occurs when the test suite is based on testing all the methods in a package, rather than units of behaviour, such as a procedure serving a web service.

References:

Quality attributes

- - Code Example
- - Cause and Effect

- - Frequency
- - Refactoring

- Design Patterns for Database API Testing 1: Web Service Saving 2 – Code

1.6.2.7 Missing Assertion Message

Definition:

- A test fails. Upon examining the output of the Test Runner, we cannot determine exactly which assertion had failed.

Code Example:

```
public void testInvoice_addLineItem7() {
    LineItem expItem = new LineItem(inv, product, QUANTITY);
    //  Exercise
    inv.addItemQuantity(product, QUANTITY);
    // Verify
    List lineItems = inv.getLineItems();
    LineItem actual = (LineItem)lineItems.get(0);
    assertEquals(expItem.getInv(), actual.getInv());
    assertEquals(expItem.getProd(), actual.getProd());
    assertEquals(expItem.getQuantity(), actual.getQuantity());
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Categorising Test Smells
- xUnit test patterns: Refactoring test code

1.6.2.8 Multiple Assertions

Definition:

- When a test method contains multiple assertion statements, it is an indication that the method is testing too much

Also Known As:

- Eager Test

Code Example:

```
public class MyTestCase extends TestCase {
    public void testSomething() {
        // Set up for the test, manipulating local variables
        assertTrue(condition1);
        assertTrue(condition2);
        assertTrue(condition3);
    }
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- JUnit Anti-patterns

1.6.2.9 Piggybacking On Existing Tests

Definition:

- On the other extreme from multiple tests is piggybacking or adding assertions to existing tests to test a distinct or new feature. The more of these you add the less descriptive your test becomes.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-Patterns In Unit Testing

1.6.2.10 Split Logic

Definition:

- When the test logic is split into several test objects and their specific assertions

Code Example:

```
public class AddToCartTests {
    @Test
    public void testAddToCart() {
```

(continues on next page)

(continued from previous page)

```
        // Test adding product to cart
    }

    @Test
    public void testCartIsEmptyAfterCheckout() {
        // Checkout process
        // Assert that cart is empty
    }
}

public class CheckoutTests {
    @Test
    public void testCheckout() {
        // Checkout process
    }
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Developer test anti-patterns by lasse koskela

1.6.2.11 Split Personality

Definition:

- A test method that attempts to test several behaviors of the tested object.

Also Known As:

- Eager Test

Code Example:

```
@Test
public void testCalculatePrice() {
    // Test scenario 1: Calculate price for customer with loyalty discount
    Customer customer = new Customer();
    customer.setLoyaltyPoints(100);
    double price = Product.calculatePrice(customer, 20.0, 10.0);
    assertEquals(18.0, price, 0.0);

    // Test scenario 2: Calculate price for customer without loyalty discount
    Customer customer2 = new Customer();
    customer2.setLoyaltyPoints(0);
```

(continues on next page)

(continued from previous page)

```
    double price2 = Product.calculatePrice(customer2, 20.0, 10.0);
    assertEquals(20.0, price2, 0.0);

    // Test scenario 3: Calculate price for product with no discount
    Customer customer3 = new Customer();
    customer3.setLoyaltyPoints(50);
    double price3 = Product.calculatePrice(customer3, 30.0, 0.0);
    assertEquals(30.0, price3, 0.0);
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Developer test anti-patterns by lasse koskela

1.6.2.12 Test Cases Are Concerned With More Than One Unit Of Code

Definition:

- When tests get too big they can have bugs of their own

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-Patterns In Unit Testing

1.6.2.13 The Free Ride

Definition:

- When an extra assertion is added in an existing test to cover a new scenario case

Also Known As:

- Eager Test

Code Example:

```
public void CalculateDiscount_ExpectedDiscountForFirstTimePurchase()
{
    //Arrange
    decimal expected = 0.1M;
    decimal expectedResultAge = 0.2M;

    var sut = new DiscountCalculator();

    //Act
    var result = sut.CalculateDiscount(true, 30);
    var resultAge = sut.CalculateDiscount(false, 65);

    //Assert
    Assert.Equal(expected, result);
    Assert.Equal(expectedResultAge, resultAge);
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-Patterns - Digital Tapestry
- Smells in software test code: A survey of knowledge in industry and academia
- TDD Anti-patterns: The Free Ride / Piggyback
- TDD and anti-patterns - Chapter 5
- Tdd antipatterns: The free ride
- Test-Driven Development: TDD Anti-Patterns
- Unit Testing Anti-Patterns, Full List
- Unit testing Anti-patterns catalogue

1.6.2.14 The Giant

Definition:

- Many assertions in a single test case

Code Example:

```
test('should setup middleware', async () => {
    const nuxt = createNuxt()
    const server = new Server(nuxt)
    server.useMiddleware = jest.fn()
    server.serverContext = { id: 'test-server-context' }
```

(continues on next page)

(continued from previous page)

```
    await server.setupMiddleware()

    expect(server.nuxt.callHook).toBeCalledTimes(2)
    expect(server.nuxt.callHook).nthCalledWith(1, 'render:setupMiddleware', server.app)
    expect(server.nuxt.callHook).nthCalledWith(2, 'render:errorMiddleware', server.app)

    expect(server.useMiddleware).toBeCalledTimes(4)
    expect(serveStatic).toBeCalledTimes(2)
    expect(serveStatic).nthCalledWith(1, 'resolve(/var/nuxt/src, var/nuxt/static)',␣
↪server.options.render.static)
    expect(server.useMiddleware).nthCalledWith(1, {
        dir: 'resolve(/var/nuxt/src, var/nuxt/static)',
        id: 'test-serve-static',
        prefix: 'test-render-static-prefix'
    })
    expect(serveStatic).nthCalledWith(2, 'resolve(/var/nuxt/build, dist, client)',␣
↪server.options.render.dist)
    expect(server.useMiddleware).nthCalledWith(2, {
        handler: {
        dir: 'resolve(/var/nuxt/build, dist, client)',
        id: 'test-serve-static'
        },
        path: '__nuxt_test'
    })

    const nuxtMiddlewareOpts = {
        options: server.options,
        nuxt: server.nuxt,
        renderRoute: expect.any(Function),
        resources: server.resources
    }
    expect(nuxtMiddleware).toBeCalledTimes(1)
    expect(nuxtMiddleware).toBeCalledWith(nuxtMiddlewareOpts)
    expect(server.useMiddleware).nthCalledWith(3, {
        id: 'test-nuxt-middleware',
        ...nuxtMiddlewareOpts
    })

    const errorMiddlewareOpts = {
        resources: server.resources,
        options: server.options
    }
    expect(errorMiddleware).toBeCalledTimes(1)
    expect(errorMiddleware).toBeCalledWith(errorMiddlewareOpts)
    expect(server.useMiddleware).nthCalledWith(4, {
        id: 'test-error-middleware',
        ...errorMiddlewareOpts
    })
})
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-Patterns - Digital Tapestry
- Categorising Test Smells
- Smells in software test code: A survey of knowledge in industry and academia
- TDD anti patterns - Chapter 1
- TDD anti-patterns - the liar, excessive setup, the giant, slow poke
- Test-Driven Development: TDD Anti-Patterns
- Unit Testing Anti-Patterns, Full List
- Unit testing Anti-patterns catalogue

1.6.2.15 The One

Definition:

- A combination of several patterns, particularly The Free Ride and The Giant, a unit test that contains only one test method which tests the entire set of functionality an object has. A common indicator is that the test method name is often the same as the unit test name, and contains multiple lines of setup and assertions.

Also Known As:

- Eager Test

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Anti-Patterns - Digital Tapestry
- Categorising Test Smells
- Test-Driven Development: TDD Anti-Patterns

1.6.2.16 The Test It All

Definition:

- Tests that break the Single Responsibility Principle.

Also Known As:

- Eager Test

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Unit testing Anti-patterns catalogue

1.6.3 Testing many units

1.6.3.1 Feature Envy

Definition:

- A macro event uses another macro component's macro events or components excessively.

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Bad smells and refactoring methods for GUI test script

1.6.3.2 Indirect Testing

Definition:

- The Test Method is interacting with the SUT indirectly via another object thereby making the interactions more complex.

Code Example:

```java
private final int LEGAL_CONN_MINS_SAME = 30;
public void testAnalyze_sameAirline_LessThanConnectionLimit()
throws Exception {
   // setup
   FlightConnection illegalConn = createSameAirlineConn( LEGAL_CONN_MINS_SAME - 1);
   // exercise
   FlightConnectionAnalyzerImpl sut = new FlightConnectionAnalyzerImpl();
   String actualHtml = sut.getFlightConnectionAsHtmlFragment( illegalConn.
↪getInboundFlightNumber(),
                     illegalConn.getOutboundFlightNumber());
   // verification
   StringBuffer expected = new StringBuffer();
   expected.append("<span class=”boldRedText”>");
   expected.append("Connection time between flight ");
   expected.append(illegalConn.getInboundFlightNumber());
   expected.append(" and flight ");
   expected.append(illegalConn.getOutboundFlightNumber());
   expected.append(" is ");
   expected.append(illegalConn.getActualConnectionTime());
   expected.append(" minutes.</span>");
   assertEquals("html", expected.toString(), actualHtml);
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- A survey on test practitioners’ awareness of test smells
- An Empirical Study into the Relationship Between Class Features and Test Smells
- An empirical analysis of the distribution of unit test smells and their impact on software maintenance
- An exploratory study of the relationship between software test smells and fault-proneness
- Are test smells really harmful? An empirical study
- Automatic generation of smell-free unit tests
- Categorising Test Smells
- Detecting redundant unit tests for AspectJ programs
- Enhancing developers’ awareness on test suites’ quality with test smell summaries
- How are test smells treated in the wild? A tale of two empirical studies
- Obscure Test
- On the Relation of Test Smells to Software Code Quality
- On the diffusion of test smells in automatically generated test code: an empirical study

- On the interplay between software testing and evolution and its effect on program comprehension
- Refactoring Test Code
- Revisiting Test Smells in Automatically Generated Tests: Limitations, Pitfalls, and Opportunities
- Scented since the beginning: On the diffuseness of test smells in automatically generated test code
- Test Smell Detection Tools: A Systematic Mapping Study
- TestQ: Exploring Structural and Maintenance Characteristics of Unit Test Suites
- The Relation of Test-Related Factors to Software Quality: A Case Study on Apache Systems
- Why do builds fail?—A conceptual replication study

1.6.3.3 Test Envy

Definition:

- A test that is really testing some other class than the one it is supposed to. A common example of this is testing model validations in the controller when they should be tested in the model test

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency
- - Refactoring

- Categorising Test Smells
- Smells of Testing (signs your tests are bad)

1.6.3.4 The Stranger

Definition:

- A test case that doesn't even belong in the unit test it is part of. it's really testing a separate object, most likely an object that is used by the object under test, but the test case has gone and tested that object directly without relying on the output from the object under test making use of that object for its own behavior. Also known as TheDistantRelative

Also Known As:

- The Cuckoo

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency

- - Refactoring

- Anti-Patterns - Digital Tapestry
- Categorising Test Smells
- Test-Driven Development: TDD Anti-Patterns

1.7 Easy Lab

We are the Engineering and Systems Laboratory (EASY), from the Federal University of Alagoas (UFAL), Brazil. Check us out! (Portuguese only) https://easy-group.netlify.app

1.8 How to contribute

First, you must know what you want to contribute.

1. Is it a new smell?
2. Have you found a bug?
3. Do you want to modify an already cataloged smell?

1.8.1 1. Contributing with a new smell

You may create a new smell page, following the *template.rst* available on this very directory, or open a new issue on our GitHub page.

1.8.2 2. Bug reporting

If you have found a bug, edit the page on GitHub or create a new issue!

1.8.3 3. I want to modify an already cataloged smell

You need to create a new pull request with the changes or email us on easy@ic.ufal.br

Tip: You may click *Edit on GitHub* in the top right corner of this page to get fast access to our GitHub repo!