Unit tests using JUnit; coverage as a quality metric

DETI-UA/TQS

Ilídio Oliveira (ico@ua.pt)

v2025-02-11

universidade de aveiro
departamento de eletrónica, telecomunicações e informática

deti

Learning objectives

Explain the concept of the “test pyramid”

Describe the 3 main layers in the test pyramid

Identify relevant unit tests for a given contract

Enumerate best practices for unit testing

Write unit test using JUnit constructions

Analyze code coverage

Discuss spectrum-based fault location techniques

Verification vs Validation

VERIFICATION: ARE WE DOING THE SYSTEM IN THE RIGHT WAY?

Check work products against their specifications
Check modules consistency
Check against industry best practices
...

CORRECTNESS

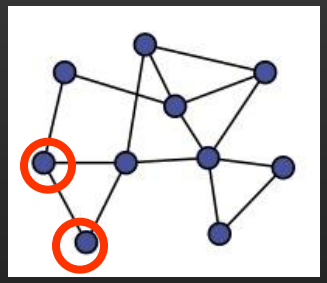
VALIDATION: ARE WE DOING THE RIGHT SYSTEM?

Check work-products against the user needs and expectations

ACCEPTANCE

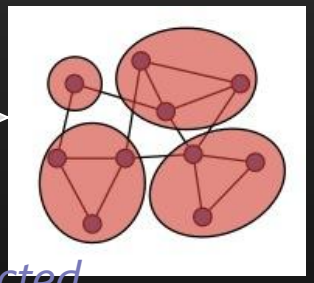
Different testing techniques are appropriate at different moments/scopes

Unit testing

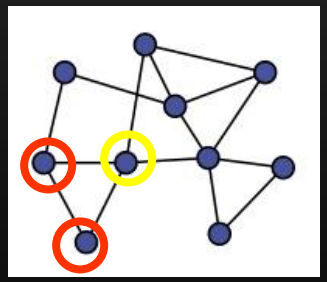


Each module does what it is supposed to do?

integration testing



managing complexity



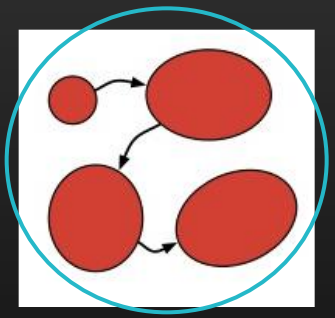
Do you get the expected results when the parts are put together?

Integration testing

Developer vs customer

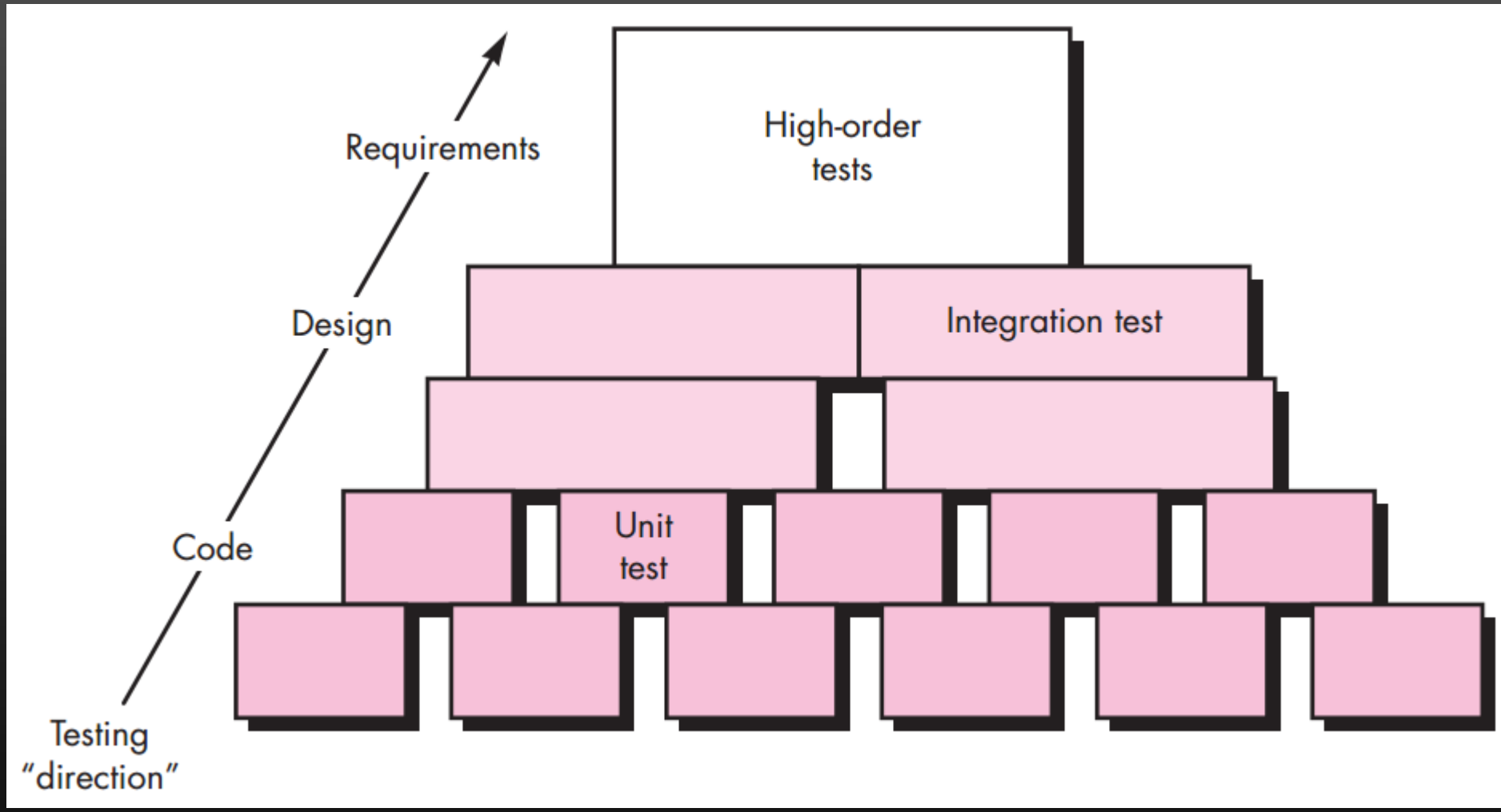
Does the program satisfy the requirements?

User Acceptance testing



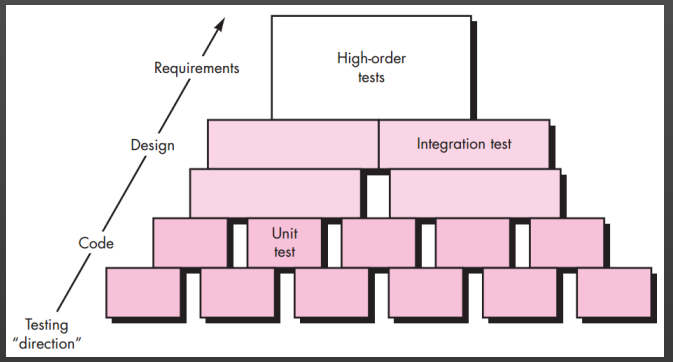
System testing

The whole system functions as expected, in the target config?



Testing begins at "celular" level and works outwards.

Which “scope”?



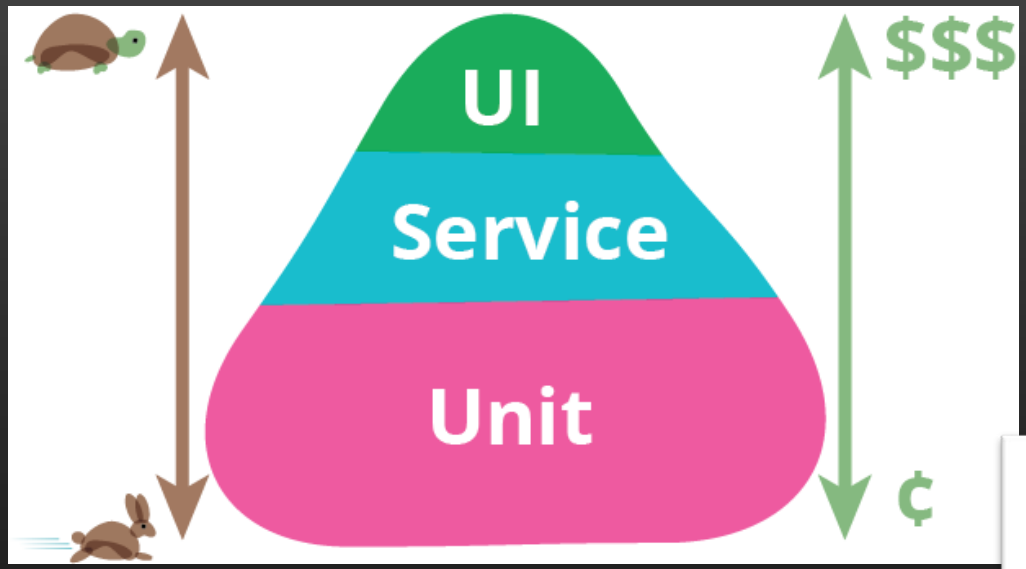
The Dice class should provide random draws.

The EmployeeManager service can list monthly top performers?

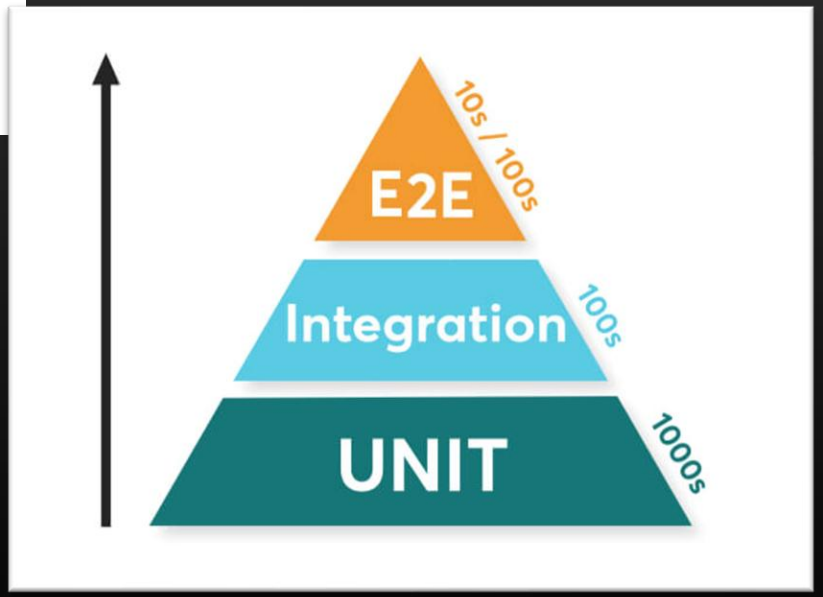
The visitor will search by free text in the product browsing page.

Is the API handling timeouts from the online micro-payment services?

The test pyramid metaphor



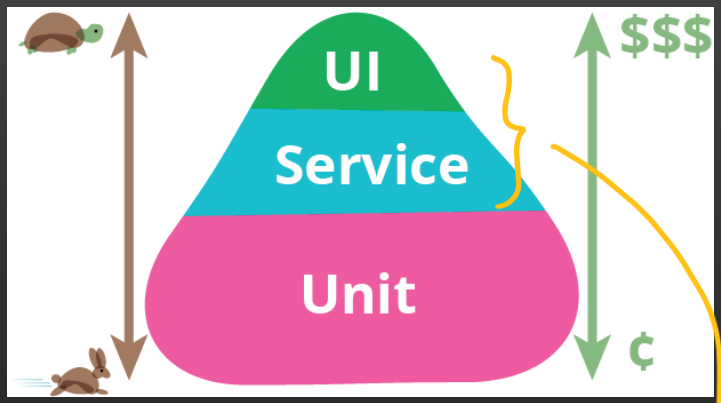
https://martinfowler.com/bliki/TestPyramid.html

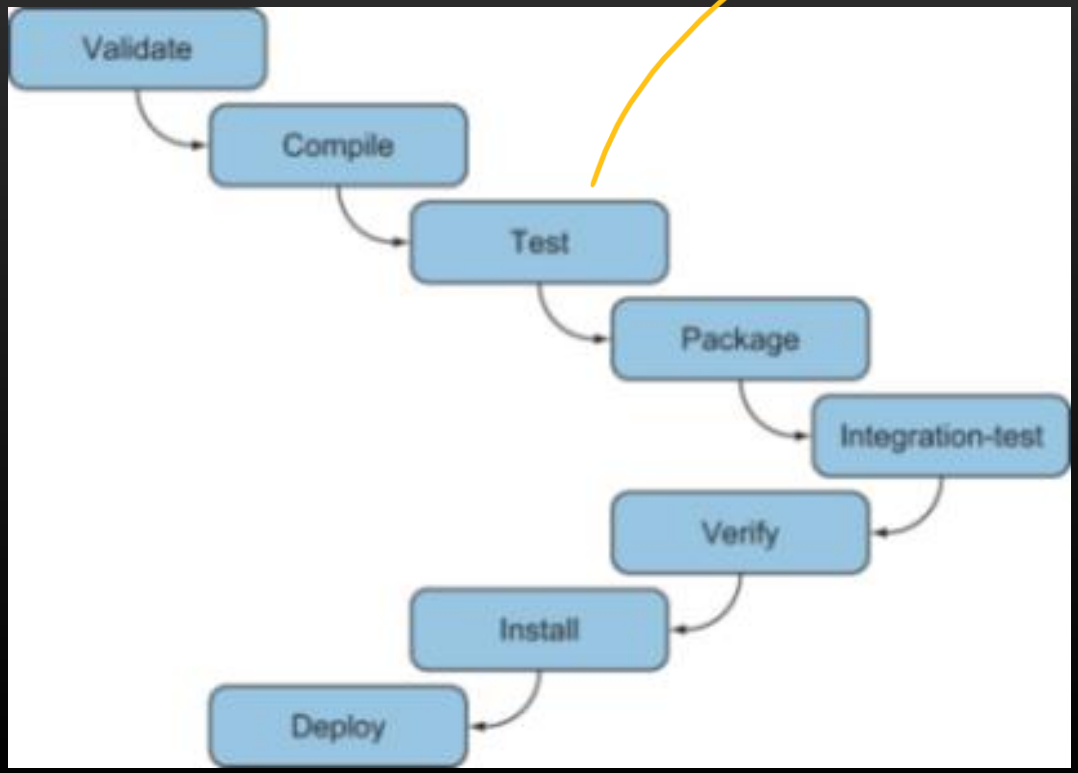


https://www.blazemeter.com/blog/agile-development-and-testing-an-introduction

Maven build cycle and tests

Maven surefire plugin





Maven failsafe plugin

Role of Unit testing

Test “components” individually

focused and concise tests

Answer the question: does the component function correctly, in isolation?

What is a “component”, in this context?

A basic build block, often a single class.

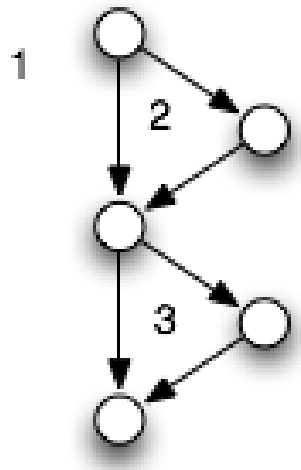
Typically implemented by a single developer.

Strategy

- Exercise different paths in a component’s control structure
- Aim at ↑ coverage
- must integrate in build tools (e.g.: Maven can run tests and report results)

Unit tests: white or black box approach?

```
def my_method (x, y)
    r = x
    if x > 5
        r = 5
    end
    if y < 5
        r = y
    end
    r
end
```

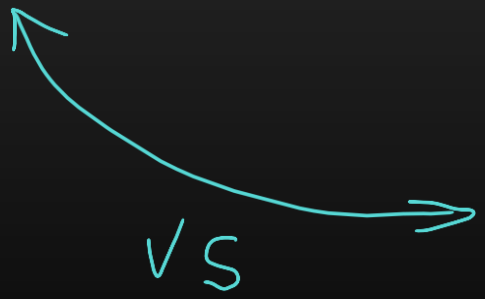


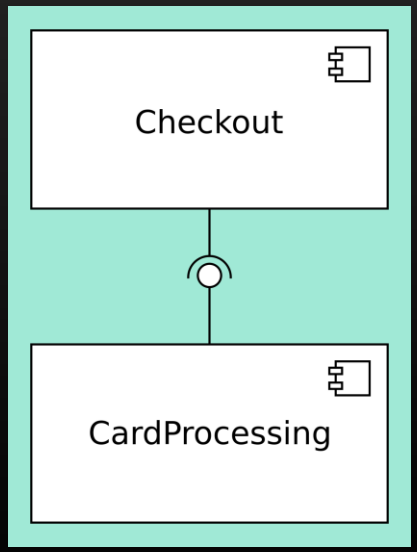
Test possible paths?

Test boundary values?

Test exceptions?

Is “inside knowledge” required?





JUnit framework

A first test case

```
import static org.junit.jupiter.api.Assertions.assertEquals;

import example.util.Calculator;

import org.junit.jupiter.api.Test;

class MyFirstJUnitJupiterTests {

    private final Calculator calculator = new Calculator();

    @Test
    void addition() {
        assertEquals(2, calculator.add(1, 1));
    }

}
```

```
assertEquals(4, calculator.multiply(2, 2),
        "The optional failure message is now the last parameter");
```

Selected annotations

https://junit.org/junit5/docs/current/user-guide/#writing-tests-annotations

JUnit Annotation	Meaning
@Test	Denotes that a method is a test method.
@BeforeEach	The annotated method should be executed *before* **each** @Test
@AfterEach	The annotated method should be executed *after* **each** @Test
@ParameterizedTest	Denotes that a method is a test method. Can provide “data” to be used in the test execution.
@DisplayName	Declares a custom display name for the test class or test method.
@Disabled	[temporarily] Disable a test class or test method
…	

Assertions

```
class AssertionsDemo {

    private final Calculator calculator = new Calculator();

    private final Person person = new Person("Jane", "Doe");

    @Test
    void standardAssertions() {
        assertEquals(2, calculator.add(1, 1));
        assertEquals(4, calculator.multiply(2, 2),
                "The optional failure message is now the last parameter");
        assertTrue('a' < 'b', () -> "Assertion messages can be lazily evaluated -- "
                + "to avoid constructing complex messages unnecessarily.");
    }

    @Test
    void groupedAssertions() {
        // In a grouped assertion all assertions are executed, and all
        // failures will be reported together.
        assertAll("person",
            () -> assertEquals("Jane", person.getFirstName()),
            () -> assertEquals("Doe", person.getLastName())
        );
    }
```

Testing for expected exceptions

```
@Test
void exceptionTesting() {
    Exception exception = assertThrows(ArithmeticException.class, () ->
        calculator.divide(1, 0));
    assertEquals("/ by zero", exception.getMessage());
}
```

Parameterized tests

```java
@ParameterizedTest
@ValueSource(strings = { "racecar", "radar", "able was I ere I saw elba" })
void palindromes(String candidate) {
    assertTrue(StringUtils.isPalindrome(candidate));
}
```

When executing the above parameterized test method, each invocation will be reported separately. For instance, the `ConsoleLauncher` will print output similar to the following.

```
palindromes(String) ✓
├─ [1] candidate=racecar ✓
├─ [2] candidate=radar ✓
└─ [3] candidate=able was I ere I saw elba ✓
```

Assertions libraries: AssertJ

```java
// entry point for all assertThat methods and utility methods (e.g. entry)
import static org.assertj.core.api.Assertions.*;

// basic assertions
assertThat(frodo.getName()).isEqualTo("Frodo");
assertThat(frodo).isNotEqualTo(sauron);

// chaining string specific assertions
assertThat(frodo.getName()).startsWith("Fro")
                           .endsWith("do")
                           .isEqualToIgnoringCase("frodo");

// collection specific assertions (there are plenty more)
// in the examples below fellowshipOfTheRing is a List<TolkienCharacter>
assertThat(fellowshipOfTheRing).hasSize(9)
                               .contains(frodo, sam)
                               .doesNotContain(sauron);
```

https://assertj.github.io/doc/

Assertions libraries: Hamcrest

The *is* construct on a simple data type:

```
@Test
public void given2Strings_whenIsEqual_thenCorrect() {
    String str1 = "text";
    String str2 = "text";
    assertThat(str1, is(str2));
}
```

To check if a *Collection* contains given members, regardless of order:

```
@Test
public void givenAListAndValues_whenChecksListForGivenValues_thenCorrect() {
    List<String> hamcrestMatchers = Arrays.asList(
      "collections", "beans", "text", "number");
    assertThat(hamcrestMatchers,
    containsInAnyOrder("beans", "text", "collections", "number"));
}
```

https://hamcrest.org/JavaHamcrest/tutorial

Keeping Tests **Consistent with AAA**

```
import static org.junit.Assert.*;
import static org.hamcrest.CoreMatchers.*;
import org.junit.*;

public class ScoreCollectionTest {
   @Test
   public void answersArithmeticMeanOfTwoNumbers() {
      // Arrange
      ScoreCollection collection = new ScoreCollection();
      collection.add(() -> 5);
      collection.add(() -> 7);

      // Act
      int actualResult = collection.arithmeticMean();

      // Assert
      assertThat(actualResult, equalTo(6));
   }
}
```

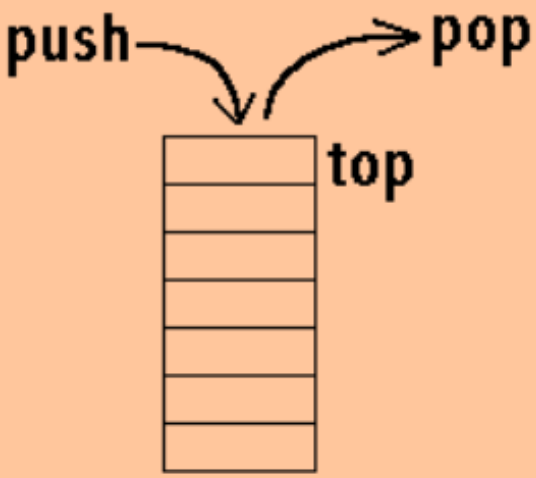
Arrange. Ensure that the system is in a proper state by creating objects, interacting with them, calling other APIs, and so on.

Act. Exercise the code we want to test, usually by calling a single method.

Assert. Verify that the exercised code behaved as expected. This can involve inspecting the return value or the new state of any objects involved. The blank lines that separate each portion of a test are a visual reinforcement to help you understand a test more quickly.

[After. If running the test results in any resources being allocated, ensure that they get cleaned up.]

Unit test: stack contract



Operations

- `push(x)`: add an item on the top
- `pop`: remove the item at the top
- `peek`: return the item at the top (without removing it)
- `size`: return the number of items in the stack
- `isEmpty`: return whether the stack has no items

Unit test example: Verifying the unit contract

a) A stack is empty on construction

b) A stack has size 0 on construction

c) After n pushes to an empty stack, n > 0, the stack is not empty && its size is n

d) If one pushes x then pops, the value popped is x, the size is decreased by one.

e) If one pushes x then peeks, the value returned is x, but the size stays the same

f) If the size is n, then after n pops, the stack is empty and has a size 0

g) Popping from an empty stack does throw a NoSuchElementException

h) Peeking into an empty stack does throw a NoSuchElementException

i) For bounded stacks only, pushing onto a full stack does throw an IllegalStateException

→ See also: Ray Toal's notes.

JUnit 5	JUnit 4	Description
`import org.junit.jupiter.api.*`	`import org.junit.*`	Import statement for using the following annotations.
`@Test`	`@Test`	Identifies a method as a test method.
`@BeforeEach`	`@Before`	Executed before each test. It is used to prepare the test environment (e.g., read input data, initialize the class).
`@AfterEach`	`@After`	Executed after each test. It is used to cleanup the test environment (e.g., delete temporary data, restore defaults). It can also save memory by cleaning up expensive memory structures.
`@BeforeAll`	`@BeforeClass`	Executed once, before the start of all tests. It is used to perform time intensive activities, for example, to connect to a database. Methods marked with this annotation need to be defined as `static` to work with JUnit.
`@AfterAll`	`@AfterClass`	Executed once, after all tests have been finished. It is used to perform clean-up activities, for example, to disconnect from a database. Methods annotated with this annotation need to be defined as `static` to work with JUnit.
`@Disabled` or `@Disabled("Why disabled")`	`@Ignore` or `@Ignore("Why disabled")`	Marks that the test should be disabled. This is useful when the underlying code has been changed and the test case has not yet been adapted. Or if the execution time of this test is too long to be included. It is best practice to provide the optional description, why the test is disabled.

Anti-pattern: don't combine test methods

One unit test equals one @Test method

If you need to use the same block of code in more than one test, extract it into a utility

if all methods can share the code, put it into the fixture.

```
@Test
public void testAddAndProcess()
{
    Request request = new SampleRequest();
    RequestHandler handler = new SampleHandler();
    controller.addHandler(request, handler);
    RequestHandler handler2 = controller.getHandler(request);
    assertEquals(handler2,handler);

    // DO NOT COMBINE TEST METHODS THIS WAY
    Response response = controller.processRequest(request);
    assertNotNull("Must not return a null response", response);
    assertEquals(SampleResponse.class, response.getClass());
}
```

Testing for add

Testing for process

Unit tests: properties of good tests

Automatic

Can be run by an automation tool (vs. interactive)

Thorough

Meets the desired coverage objectives (complete, careful)

exercise the expected as well as the boundary/exceptional conditions

Repeatable

able to be run repeatedly and continue to produce the same results, regardless of the environment (vs. hard-coded URL or IDs)

Independent

Not depend or interfere with other tests

- you cannot rely upon one unit test to do the setup work for another unit test

not guaranteed to run in a particular order

What not to test

- Getters and setters
- Framework code
  - Specially generated code
- Same conditions
  - No point in having multiple tests for the same behavior or conditions
- Complex behavior from collaborating objects
  - that is not unit testing!

Code coverage

Code coverage as a quality metric

- measures the extent to which the source code of a program is executed/exercised when a particular test suite runs.
- helps developers assess test effectiveness, identify untested parts of the code

For **statement coverage**, the formula would be:

$$\text{Statement Coverage}(\%) = \left(\frac{\text{Number of Executed Statements}}{\text{Total Number of Statements}}\right) \times 100$$

Implications of high/low values of coverage in the software reliability?

Different criteria:

Statement Coverage – Measures the percentage of executed statements.

Branch Coverage – Measures whether all possible branches (e.g., if and else conditions) have been executed.

Function Coverage – Measures whether all functions or methods have been called.

Path Coverage – Measures whether all possible execution paths have been traversed.

Condition Coverage – Ensures that each boolean condition in decision-making constructs has been evaluated to both true and false.

Fault location techniques

FL: software debugging techniques to help developers understand which parts of the code are most likely responsible for causing test failures.

Spectrum-Based Fault Localization (SBFL) techniques use execution traces of test cases (=spectrum, as generated by Jacoco) to determine how likely specific program elements (e.g., statements) are responsible for observed failures.

SBFL techniques analyze **how frequently different parts of the code participate in passing and failing test cases**. By comparing the execution patterns, SBFL **ranks code elements** based on their likelihood of being faulty.

SBFL procedure:

1.Run the Test Suite:

2.Collect Execution Data: Track which statements or branches were executed during each test.

3.Compute Suspiciousness Scores: Use heuristics to compute how "suspicious" each

4.Rank Code Elements

5.Debugging and Fixing: investigate the most suspicious code elements first.

SBFL heuristic: Tarantula

T. assigns a **suspiciousness score** to each statement based on how often it is executed in **failing** vs. **passing** tests.

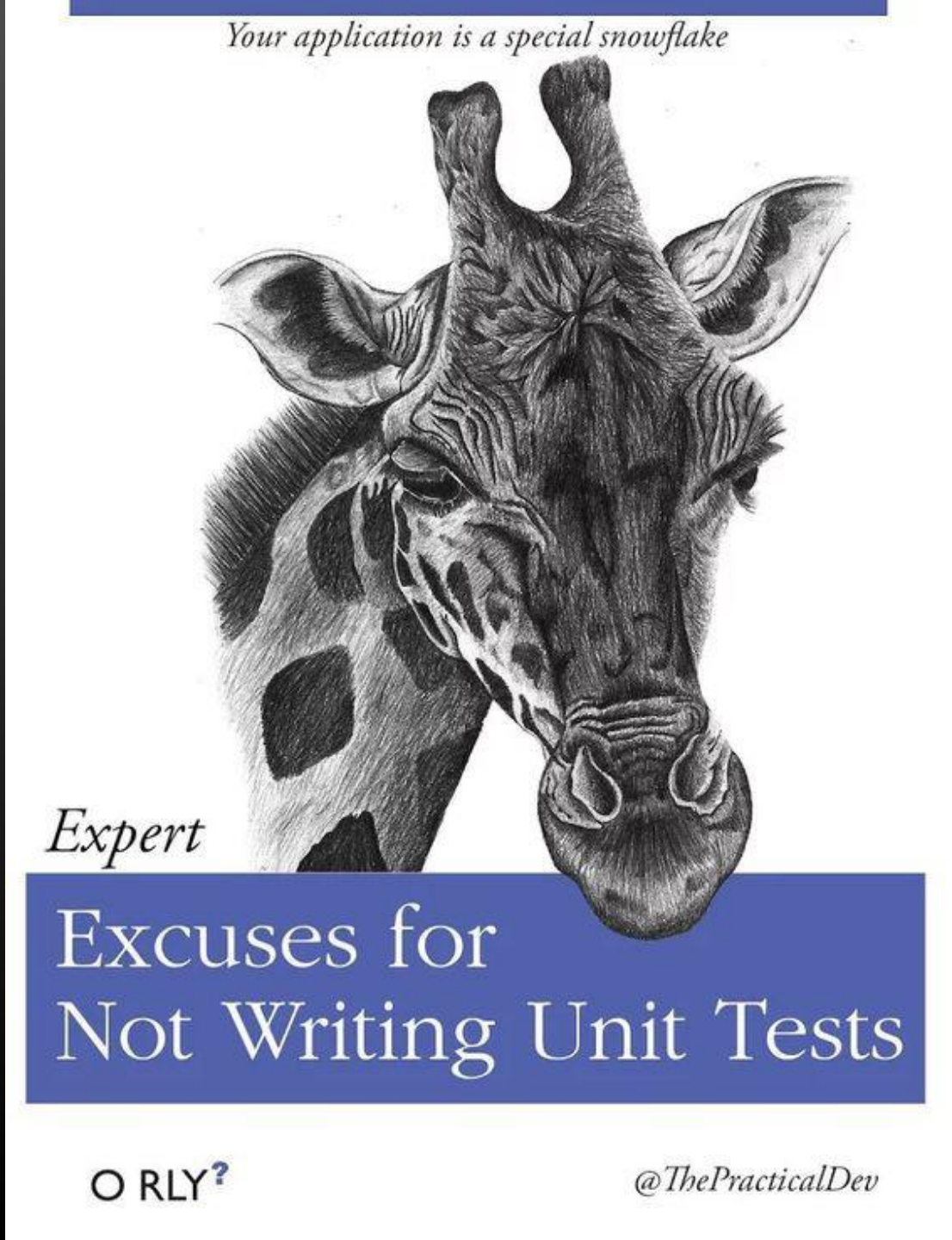
One of the most used in software methods research for benchmarking

For a statement s, the suspiciousness score is calculated as:

$$\text{Suspiciousness}(s) = \frac{\frac{e_f(s)}{t_f}}{\frac{e_f(s)}{t_f} + \frac{e_p(s)}{t_p}}$$

where:

- $e_f(s)$ = Number of **failing** tests that execute statement s.
- $e_p(s)$ = Number of **passing** tests that execute statement s.
- t_f = Total number of **failing** tests.
- t_p = Total number of **passing** tests.


Your application is a special snowflake
Expert
Excuses for
Not Writing Unit Tests
O RLY?
@ThePracticalDev

References

P. Tahchiev, F. Leme, V. Massol, and G. Gregory, JUnit in Action, Second Edition. Manning Publications, 2010.

Langr, J., Hunt, A. and Thomas, D., 2015. *Pragmatic Unit Testing in Java 8 with JUnit*. Pragmatic Bookshelf.

Stack implementation with tests:

http://cs.lmu.edu/~ray/notes/stacks/