Unit tests using JUnit

DETI-UA/TQS

Ilídio Oliveira (ico@ua.pt)

v2022-03-08

universidade de aveiro
departamento de eletrónica, telecomunicações e informática

deti

Learning objectives

- Explain the concept of the “test pyramid”
- Enumerate and distinguish 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

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

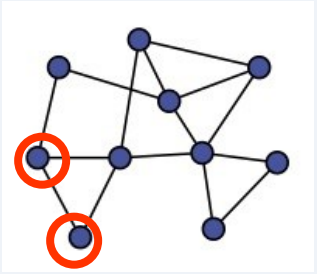
...

VALIDATION: ARE WE DOING THE RIGHT SYSTEM?

Check work-products against the user needs and expectations

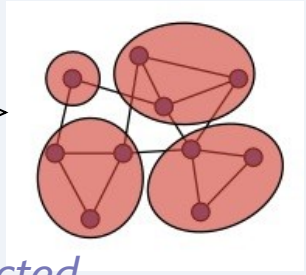
Different testing techniques are appropriate at different moments/scopes

Unit testing

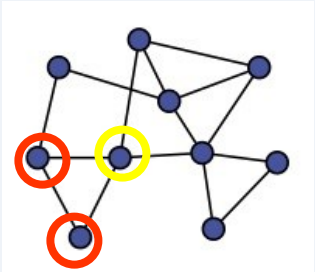


Each module does what it is supposed to do?

integration testing



managing complexity



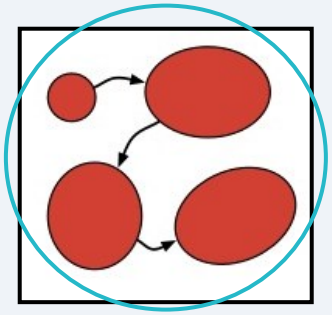
Do you get the expected results when the parts are put together?

Integration testing

Developer vs customer

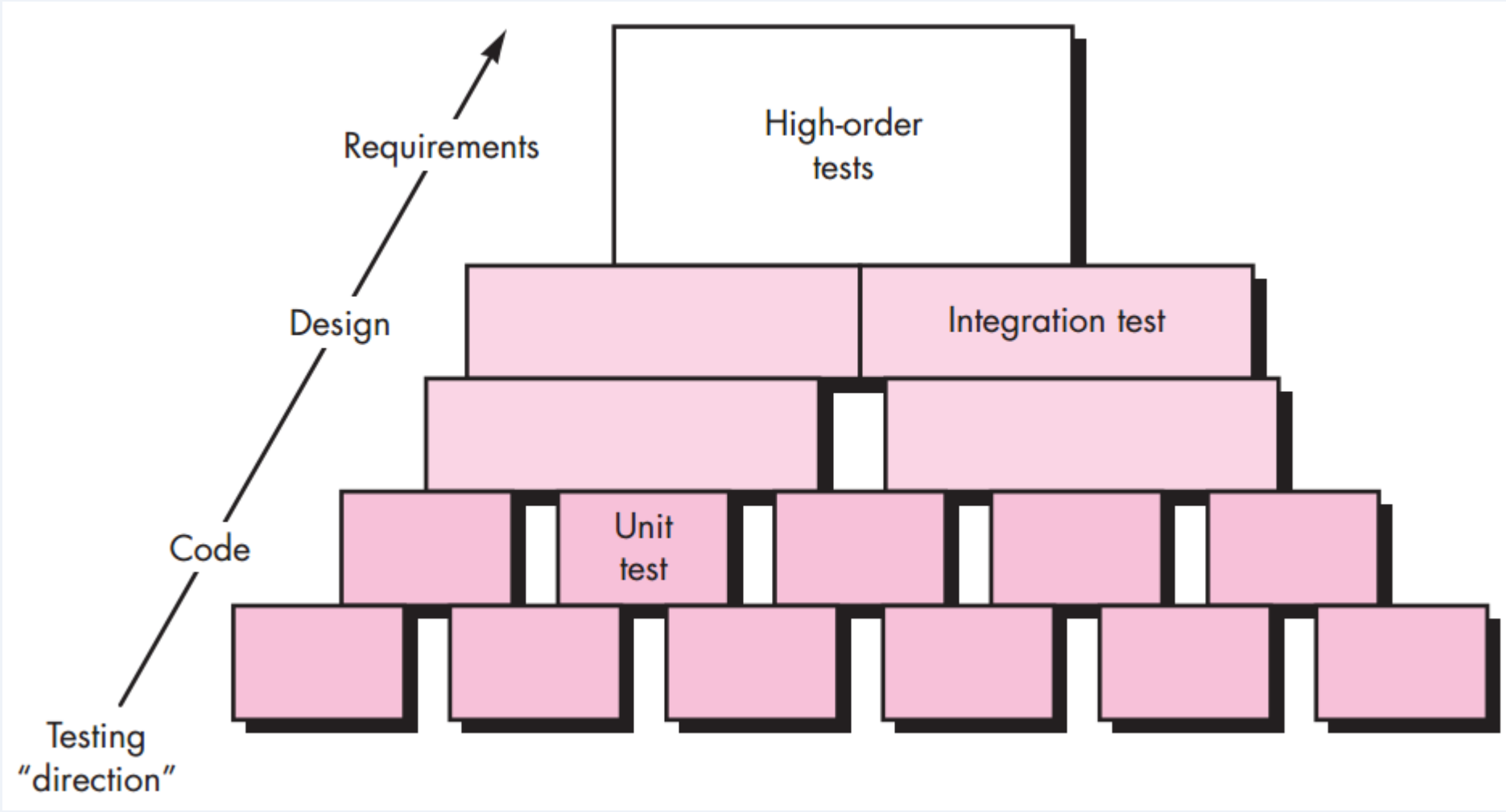
Does the program satisfy the requirements?

User Acceptance testing



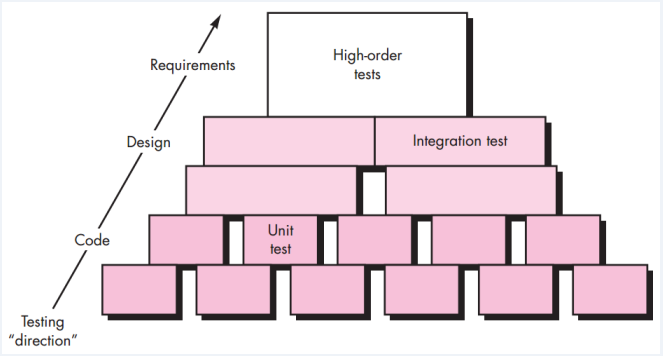
System testing

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



Testing begins at "celular" level and works outwards.

Which “scope”?



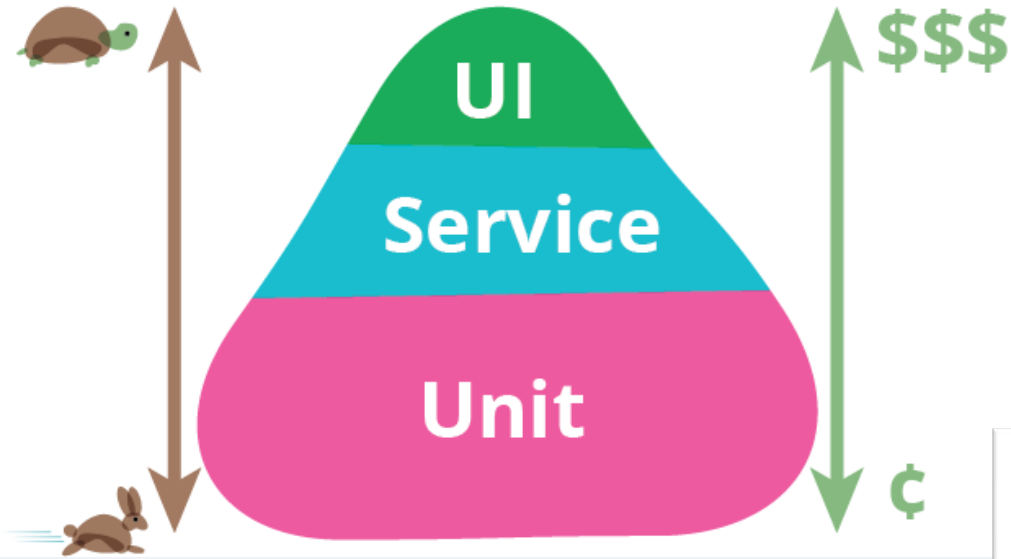
The Dice class should provide random draws.

The EmployeeManager service can list monthly top performers?

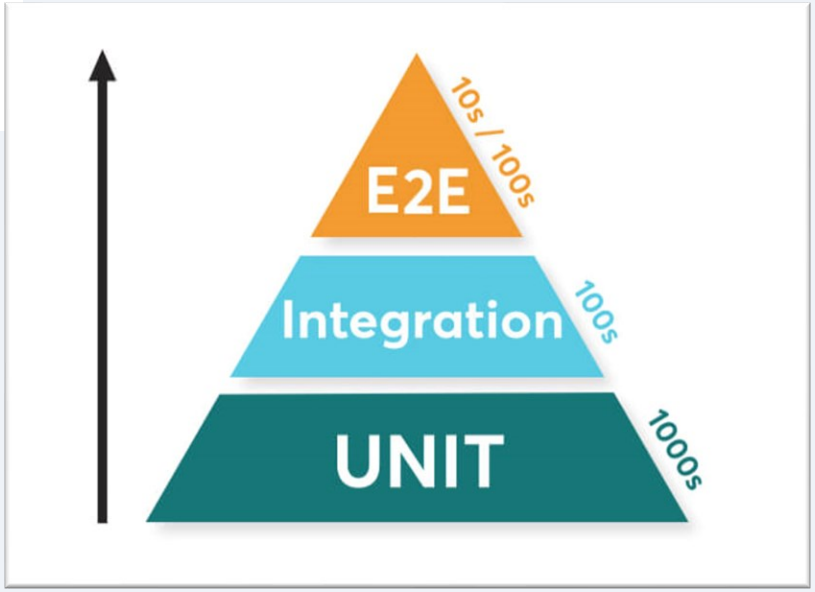
The visitor will search by free text in the product browsing page.

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

Unit testing purpose

Test “components” individually

- focused and concise tests
- Answer the question: does the component function properly, in isolation?

What is a “component”, in this context?

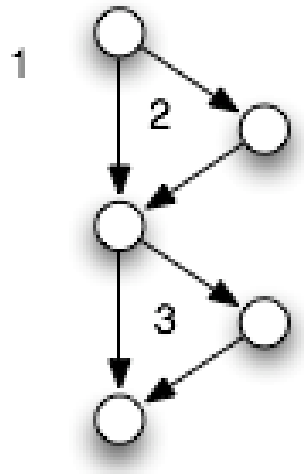
- A basic build block, often a single class.
- Typically implemented by a single developer.

Strategy

- Heavy use of testing techniques that exercise different paths in a component’s control structure
- ↑ coverage
- must integrate in build tools (e.g.: Maven can run tests and report results)

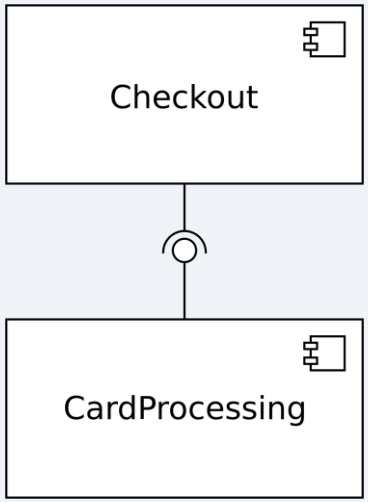
Unit tests: white or black box approach?

```
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 ✓
```

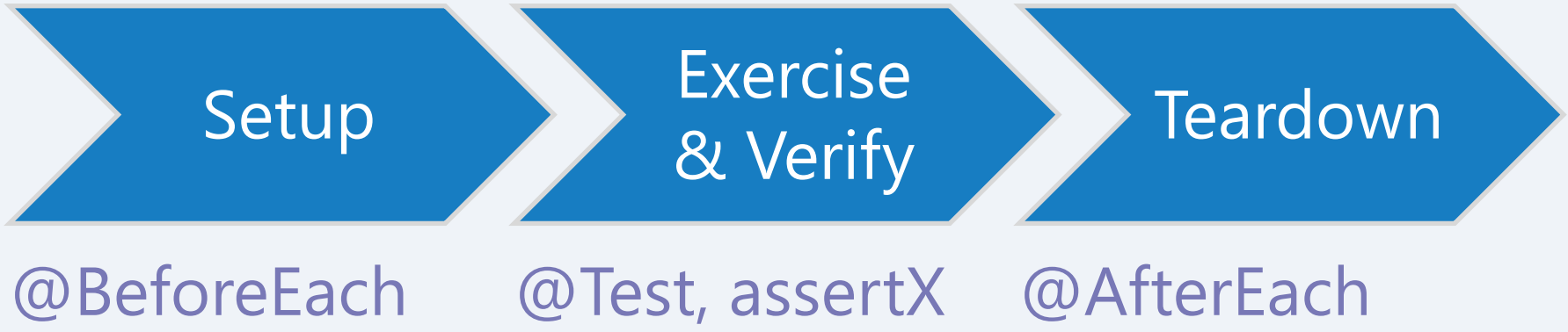
Summing up

JUnit creates a new instance of the test class before invoking each @Test method

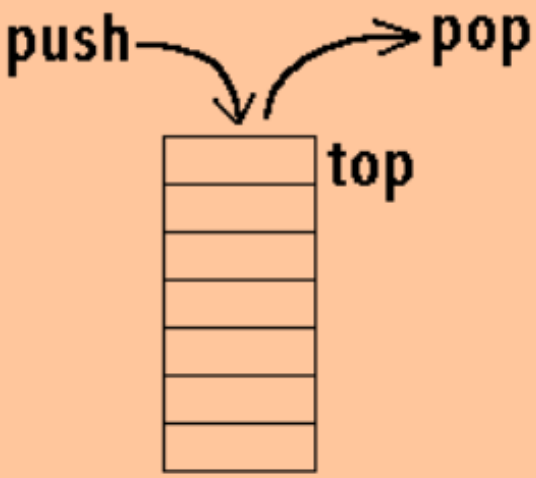
Assert methods

- assertXXX(useful failure explanation, expected, obtained)

Typical 3 phases:



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.

http://www.vogella.com/tutorials/JUnit/article.html#junit_links

More JUnit

DEFINITIONS *Test class (or TestCase or test case)*—A class that contains one or more tests represented by methods annotated with `@Test`. Use a test class to group together tests that exercise common behaviors. In the remainder of this book, when we mention a *test*, we mean a method annotated with `@Test`; when we mention a test case (or test class), we mean a class that holds these test methods—a set of tests. There's usually a one-to-one mapping between a production class and a test class.

Suite (or test suite)—A group of tests. A test suite is a convenient way to group together tests that are related. For example, if you don't define a test suite for a test class, JUnit automatically provides a test suite that includes all tests found in the test class (more on that later). A suite usually groups test classes from the same package.

Runner (or test runner)—A runner of test suites. JUnit provides various runners to execute your tests. We cover these runners later in this chapter and show you how to write your own test runners.

Best practices

A key aspect of unit tests is that they're fine-grained. A unit test independently examines each object

- When an object interacts with other complex objects, you can surround the object under test with predictable test objects.
- assertXXX(expected, actual, *useful message on fail*)
- choose meaningful test method names

testXXXYYY scheme

- XXX: domain method under test
- YYYY: how the test differs (use if XXX repeats)

```
@Test public void testProcessRequestAnswers_ErrorResponse()
@Test public void testPop_AllValuesInStack()
```

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.

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 unexpected 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 several test for the same behavior or conditions
- **Complex behavior from collaborating objects**
  - Other kind of tests to handle integration

Your application is a special snowflake

Expert

Excuses for Not Writing Unit Tests

O RLY?

@ThePracticalDev

Reasons “not to be a giraffe”… (or: advantages of having unit tests)

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/