Unit tests using JUnit

DETI-UA/TQS

Ilídio Oliveira (ico@ua.pt)

v2020-02-11

universidade de aveiro
departamento de eletrónica, telecomunicações e informática

deti

Learning objectives

Explain when to write unit test

Identify relevant unit tests for a given contract

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

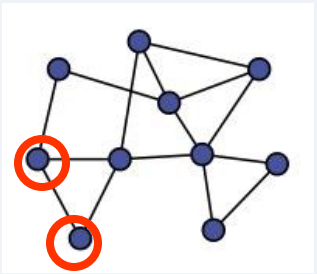
...

VALIDATION: ARE WE DOING THE RIGHT SYSTEM?

Check work-products against the user needs and expectations

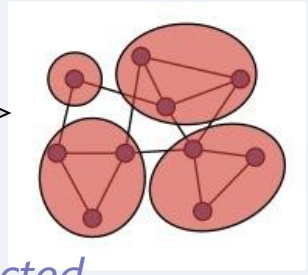
Different testing techniques are appropriate at different moments/software

Unit testing

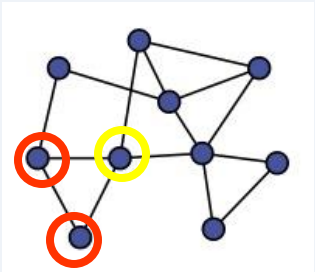


Each module does what it is supposed to do?

integration testing



managing complexity



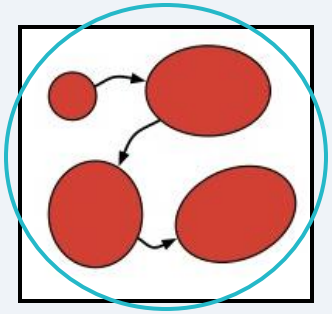
Do you get the expected results when the parts are put together?

Integration testing

Developer vs customer

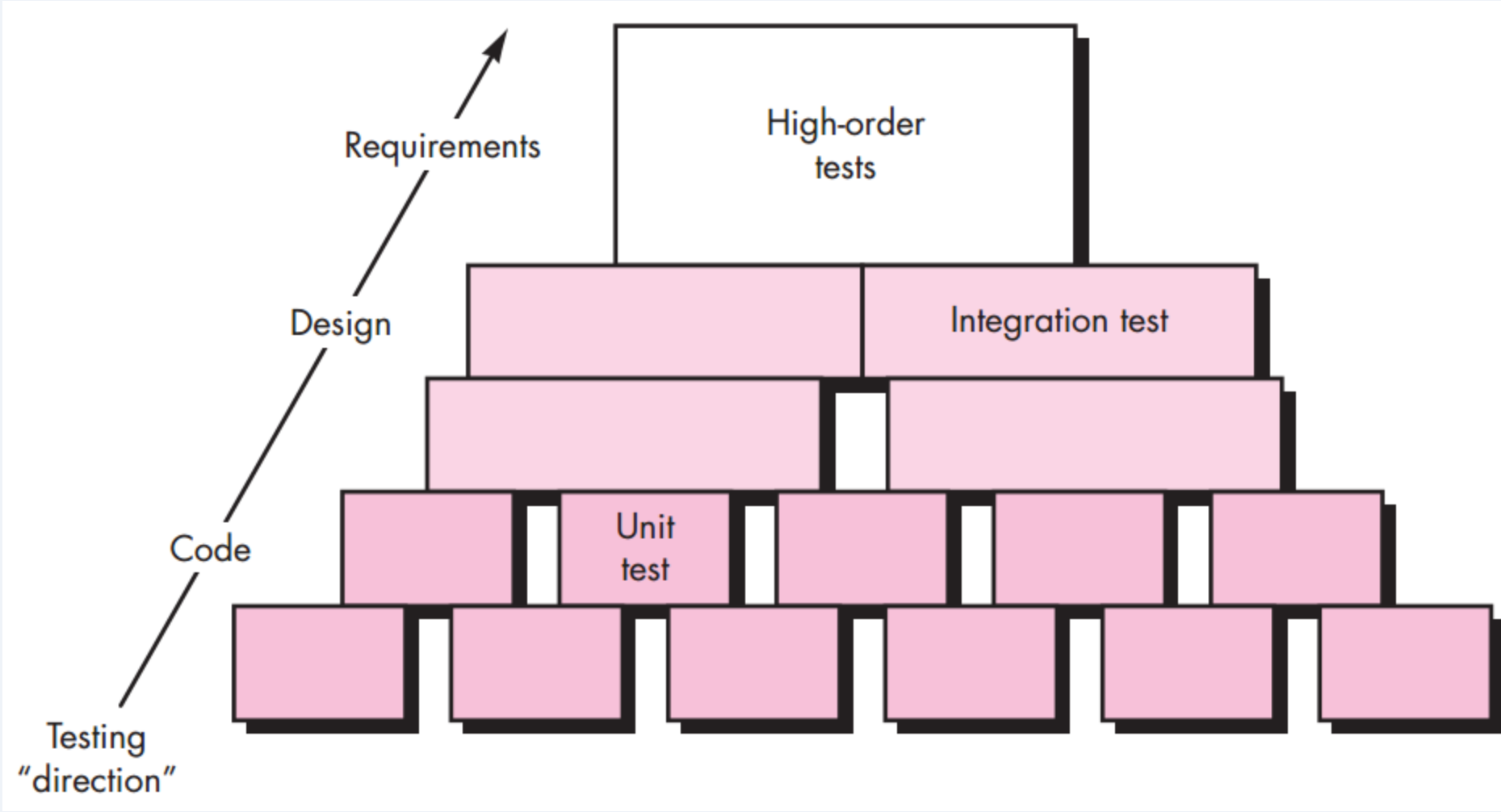
Does the program satisfy the requirements?

User Acceptance testing



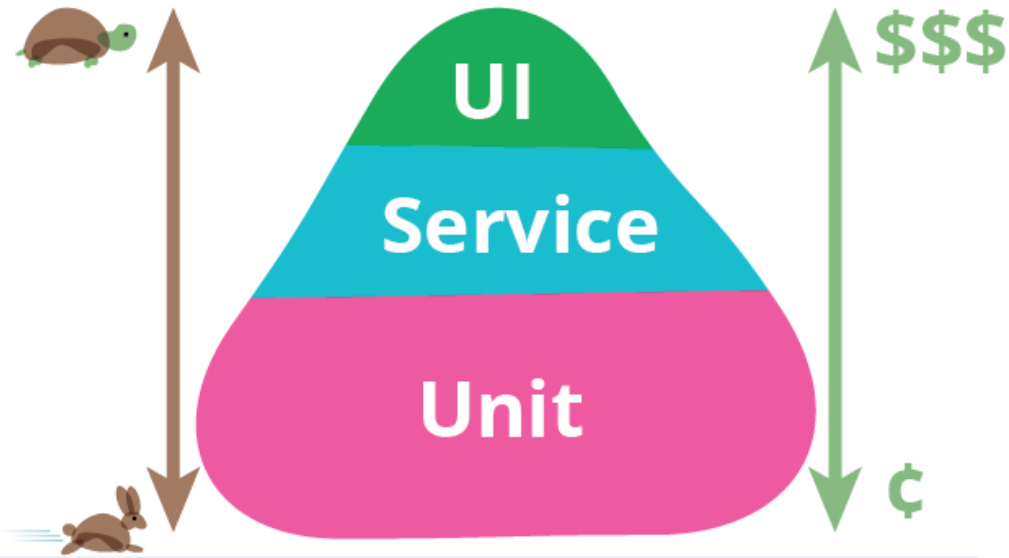
System testing

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

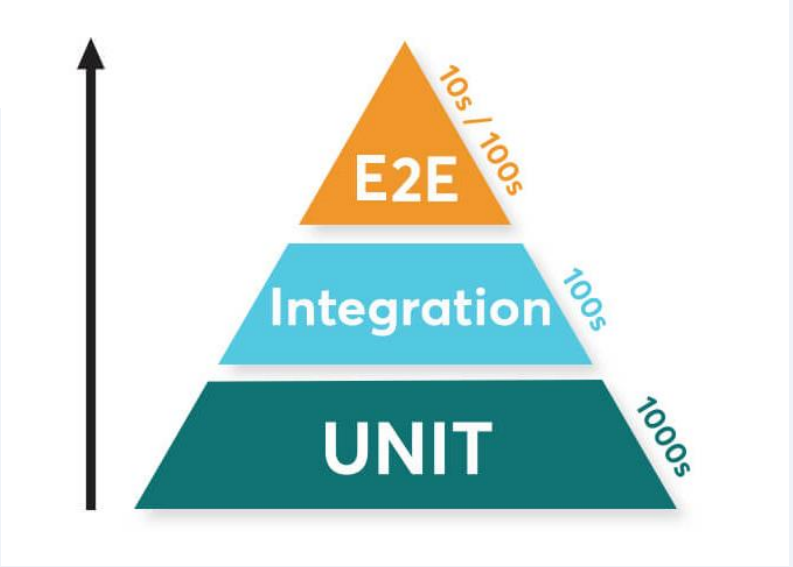


Testing begins at component level and works outwards.

The testing pyramid



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



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

Unit testing purpose

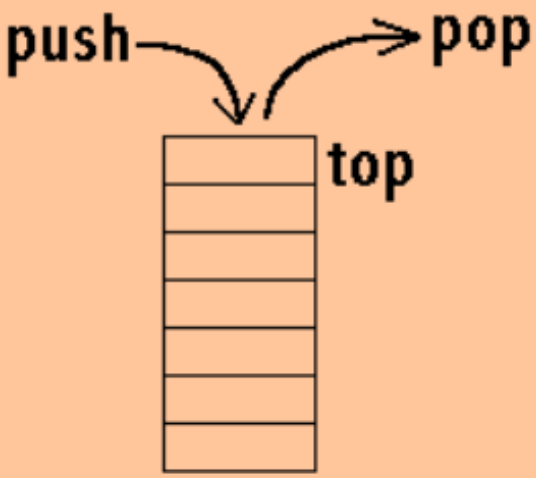
Tests components individually

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

Strategy

- Heavy use of testing techniques that exercise different paths in a component's control structure
- ↑ coverage
- must integrate in build tools

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 testing framework

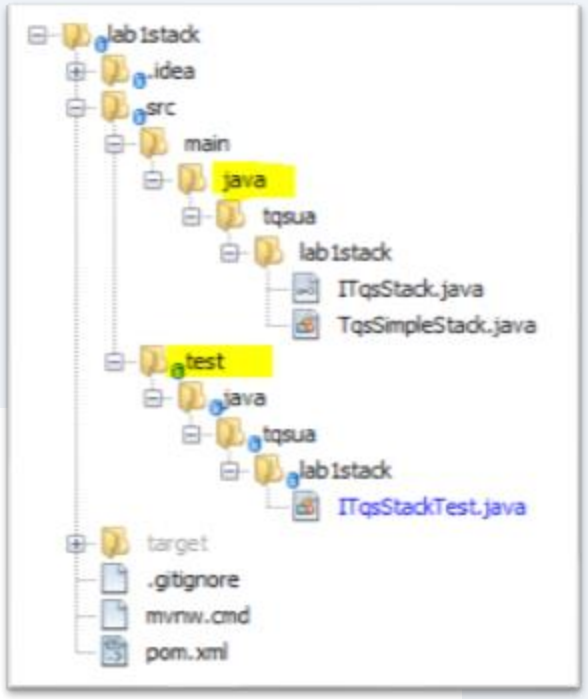
Unit test framework for Java, nicely integrated with IDEs and build tools.

http://junit.org/

JUnit in brief:

- Separate instances for each unit test to avoid side effects
- JUnit annotations to provide resource initialization and reclamation methods: @Before, @BeforeAll, @After, and @AfterAll
- A variety of assert methods
- Integration with popular build tools (e.g.: Ant, Maven) and popular IDEs (e.g.: Eclipse, NetBeans, IntelliJ)

Unit test anatomy (with Junit 4)




```
import static org.junit.Assert.*;
import org.junit.Test;

public class CalculatorTest {

    @Test
    public void testAdd() {
        Calculator calculator = new Calculator();
        double result = calculator.add(10, 50);
        assertEquals(60, result, 0);
        }
}
```

Unit test example (with Junit 5)

```java
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));
    }

}
```

JUnit workflow 1: **simple test**

Create tests with @Test annotation in testing classes

```
@Test
public void addTwoIntegersTest() {
        assertEquals("Error: wrong outcome adding 7+5!", 12,
simpleMath.add(7, 5));
}
```

assertXXX([message,] expected, exercised)

```
@Test
public void subtractionTest() {
        assertEquals(9, simpleMath.substract(12, 3));
}
```

Assert class

Table 2.1 JUnit `assert` method sample

`assertXXX` method	What it's used for
`assertArrayEquals("message", A, B)`	Asserts the equality of the `A` and `B` arrays.
`assertEquals("message", A, B)`	Asserts the equality of objects `A` and `B`. This `assert` invokes the `equals()` method on the first object against the second.
`assertSame("message", A, B)`	Asserts that the `A` and `B` objects are the same object. Whereas the previous `assert` method checks to see that `A` and `B` have the same value (using the `equals` method), the `assertSame` method checks to see if the `A` and `B` objects are one and the same object (using the == operator).
`assertTrue("message", A)`	Asserts that the `A` condition is true.
`assertNotNull("message", A)`	Asserts that the `A` object isn't null.

JUnit Workflow 2: **fixtures**

Fixture: configuration for well known object state

```
@Before
public void runBeforeEveryTest() {
        simpleMath = new SimpleMath();
}

@Test

//…

@After
public void runAfterEveryTest() {
        simpleMath = null;
}
```

S.N.	Annotation & Description
1	**@Test** The Test annotation tells JUnit that the public void method to which it is attached can be run as a test case.
2	**@Before** Several tests need similar objects created before they can run. Annotating a public void method with @Before causes that method to be run before each Test method.
3	**@After** If you allocate external resources in a Before method you need to release them after the test runs. Annotating a public void method with @After causes that method to be run after the Test method.
4	**@BeforeClass** Annotating a public static void method with @BeforeClass causes it to be run once before any of the test methods in the class.
5	**@AfterClass** This will perform the method after all tests have finished. This can be used to perform clean-up activities.
6	**@Ignore** The Ignore annotation is used to ignore the test and that test will not be executed.

JUnit workflow 3: **expecting exceptions**

```
@Test(expected = ArithmeticException.class)
public void divisionWithException() {
        // divide by zero
        simpleMath.divide(1, 0);
}

@Test(timeout = 1000)
public void testVerySlowPrint() {
        messageUtil.printMessage();
}
```

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.

RunWith... (v4)

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
@TransactionConfiguration
@Transactional
public class PersonDaoTransactionUnitTest ex

    final Logger logger = LoggerFactory.getL

    protected static int SIZE = 2;
    protected static Integer ID = new Intege
    protected static String FIRST_NAME = "Jo
    protected static String LAST_NAME = "Smi
    protected static String CHANGED_LAST_NAM

    @Autowired
```

```
import org.kiy0taka.dbunit.TestConnectio

@RunWith(DbUnitRunner.class)
public class EmpDaoTest {

    @TestConnection
    private Connection connection;

    private EmpDao dao;

    @Before
```

```
@RunWith(AnnotatedEmbedderRunner.class)
@Configure(storyLoader = MyStoryLoader.class, storyReporterBuilder = M
        parameterConverters = { MyDateConverter.class })
@UsingEmbedder(embedder = Embedder.class, generateViewAfterStories = t
@UsingSteps(instances = { TraderSteps.class, BeforeAfterSteps.class, A
        PriorityMatchingSteps.class, SandpitSteps.class })
public class TraderAnnotatedEmbedder implements Embeddable {

    private Embedder embedder;

    public void useEmbedder(Embedder embedder) {
        this.embedder = embedder;
```

```
@RunWith(MockitoJUnitRunner
public class AuctionMessage
        @Mock AuctionEventL
        @InjectMocks AuctionMessageTranslator translator = new

        @Test
        public void shouldSendAnEventT
                translator.sendMessage
```

```
13 /**
14  * @author Christopher Bartling, Pintail Consulting
15  * @since Sep 4, 2008 12:08:38 AM
16  */
17 @RunWith(JMock.class)
18 public class PricingServiceTests {
19
20     private static final String SKU = "3283947";
21     private static final String BAD_SKU = "-9999993
22
23     private PricingService systemUnderTest;
24     private DataAccess mockedDependency;
```

```
1  package com.c0deattack.cucumberjvmtutorial;
2
3  import org.junit.runner.RunWith;
4  import cucumber.junit.Cucumber;
5
6  @RunWith(Cucumber.class)
7  @Cucumber.Options(format={"pretty", "html:target/cucumber"})
8  public class RunTests {
9  }
```

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(useful message on fail, expected, actual)
- choose meaningful test method names

testXXXYYY scheme

- XXX: domain method under test
- YYYY: how the test differs (use if XXX repeats)
- @Test public void testProcessRequestAnswersErrorResponse()

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
- **Complex behavior**
  - Other kind of tests to handle integration

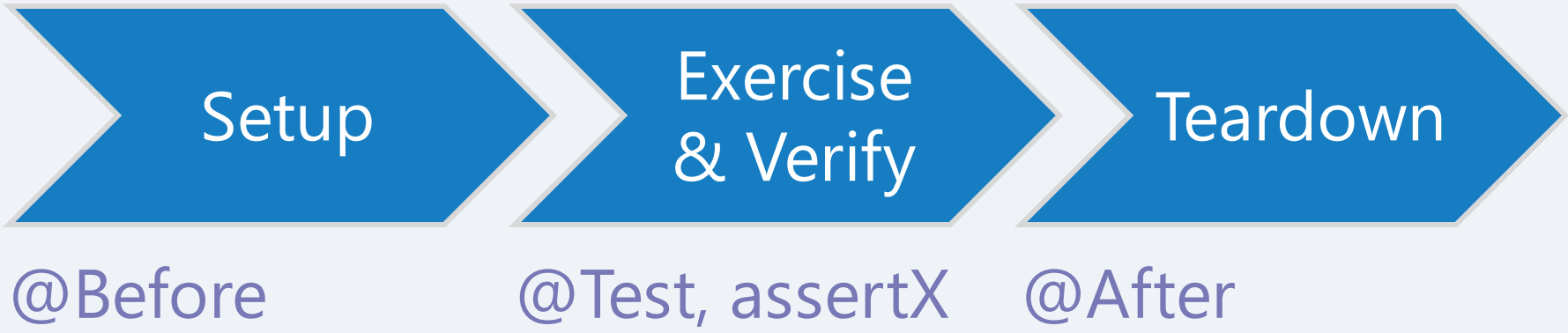
Summing up

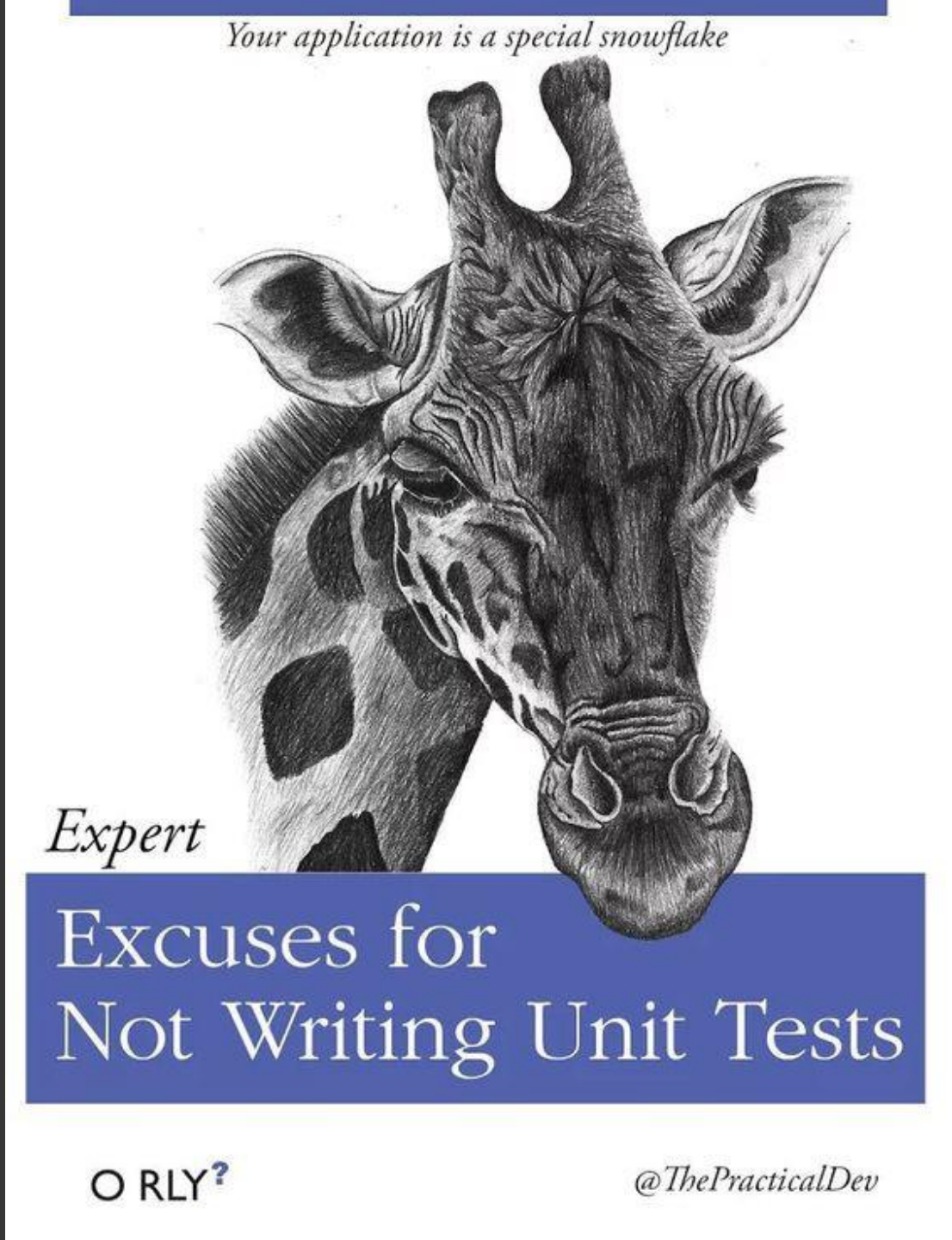
JUnit creates a new instance of the test class before invoking each @Test method

Assert methods

- assertXXX(useful failure explanation, expected, obtained)

Typical 3 phases:





https://medium.com/@gsari/15-books-that-every-programmer-should-buy-85525b509633

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/