BECA Java

everis

an NTT DATA Company

Enrique Mingorance Cano

enrique.Mingorance.cano@everis.com

Java 8

Conceptos Básicos

Tipos de Datos

Operadores y Sentencias

API Núcleo

Métodos Encapsula-ción

Excepcio nes

01

Basics

Java
Testing

Basics

Goals

- Finding the answer to the question ¿Why do we have to test?
- Learning Test best practices
- Ensuring knowledge about unit and integration testing
- Showing possible strategies in unit and integration testing
- Learning tools used in unit and integration testing

Why do we need to test?

IT SAVES TIME!!!

What does Test in Software Engineering mean?

- It’s a process which validates that software meets Functional and Non functional requirements.
- Functional Requirement is a functionality that system must do. Is the answer to the question “what”
- Non Functional Requirement is a feature that system must accomplish with. Is the answer to the question “how”.

Tests types

- Unit Test: It proves the correct behaviour of a software module.
- Integrated Test: It proves that several software modules interact right.
- Regression Test: it proves that changes performed on some system don’t have any adverse effect

Unit Test: Features

- Unitary: One test per execution path
- Automatable: Human intervention is not required.
- Complete: Unit tests should cover the largest part of the code.
- Repetibles or Reusable: Unit Tests must be runable whenever it is required, no matter how many times they have been run before.
- Independents: One execution must not affect to another one
- Proffesional: Tests must be considered as important as the code.
- Context independent: An Unit Test must not be affected by the context. A DAO should access to a data base in memory, a file reading should not access to the underlying file system…

Test Strategies

- Black Box: These are tests to prove that every functionality works without knowing how the functionality works
- White box: these are tests to prove that the inner operation meets the specifications.
- Tests must be defined to cover the highest number of possible execution paths.

Tools

- Junit: Java main unit test framework
- Mockito: Framework for objects mocking
- Maven: Software Life Cycle Tool
- eclEmma: Code cover measuring plugin for Eclipse
- PowerMockito: Framework for objects mocking. It complements Mockito when testing static methods, constructor methods, private methods…

02

JUnit

Java Testing

JUnit

2 main components

- Test Suit
- Test Case

Test Suite

- A Test Suit is a Bundle of Test Cases.
- It has NOT any code, is has just references to test cases:

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)
@SuiteClasses({Test1.class,
Test2.class})
public class AllTests {}
```

Test Case

- A Test Case has test cases.
- Test Cases should be related to the same functionality:
    - Customer management
    - Java Class
    - DAO
    - …

Test Case

Elements inside a TestCase

@RunWith: Class Annotation that references the Test Runner. Not required

@BeforeClass: Method Annotation that indicates that the method should be executed only once and before any test. Not required

@Before: Method Annotation that indicates that the method should be executed before every test. Not required

@After: Method Annotation that indicates that the method should be executed after every test. Not Required

@AfterClass: Method Annotation that indicates that the method should be executed only once and after all tests. Not required

@Test: Method Annotation that indicates that the method is a test. Required

everis
an NTT DATA Company

Unitary Test Structure

Every test method should test a single execution path inside the functionality.

No conditional structures are allowed inside @Test methods (If, If-else, Switch)

@Test method structure:

- Test Initialization //Arrange
- Functionality execution//Execute
- Result Analysis //Assert

Unitary Test Initialization

Test initialization involves

Initialization data creation

Mocks configuration

Initialization of the functionality to be tested

Any specific thing that is required for the test to work

Unitary Test Execution

Functionality execution involves

Invokating the functionality with required data

Try-catch blocks or adding “throws SomeException” to the signature could be required for the test to work.

If there is a result, this one must be recovered

Unitary Test Results

Result analysis I

When a method gives back a data structure, the data contained inside the structure must be verified. For this purpose there is a tool called Asserts.

An Assert is a condition that must be fulfilled, otherwise the test will fail.

If a method must create a file, this creation must be verified. (**Keep in mind the Context Independence**)

If data in a data base has been modified, this update must be verified against the data base (**Keep in mind the Context Independence and Repetible feature**)

Unitary Test Results

Result analysis II

If method throws an exception, this one must be caught via try-catch block or using “expected” in @Test:
@Test(expected=MyException.class)

Some points regarding to exception analysis:

- FORBIDDEN to use Exception. Use always the exception that method really throws. (In the signature could appear “throws Exception” as well. This is an error!!)
- Don’t use “expected” if it is required to make asserts over information coming from the exception itself: Codes, Description, Subtype… For this use a try-catch block and use Asserts inside the catch block.

03

Mockito

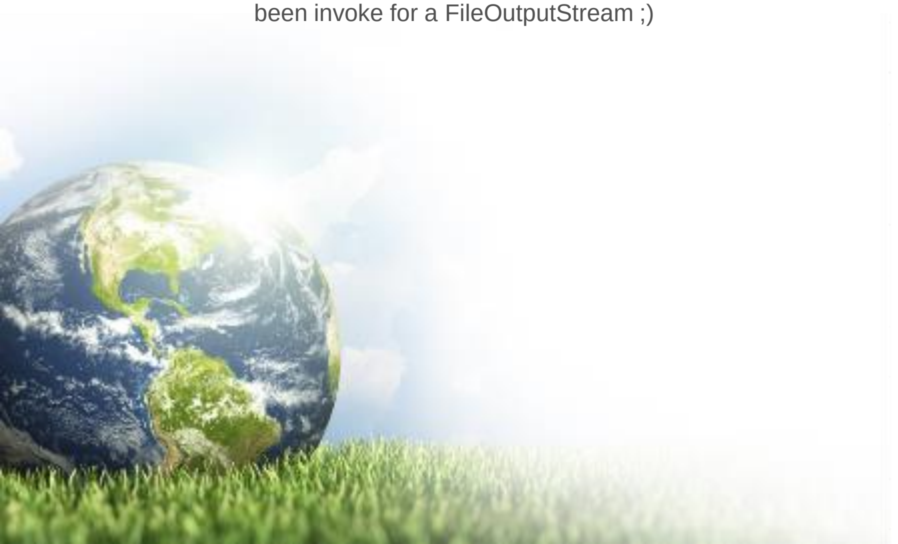
Mockito

It’s a Framework to Mock objects.

Mock means: **Imitar**

Useful when:

- You don’t have the object implementation or you want to isolate your test from the rest of the object (**Keep in mind the Unitary feature**)
- You need to model a behaviour
- You need an extra control over what has been invoked, how many times…for instance, imagine you need to control when a “close()” method has been invoke for a FileOutputStream ;)



Mockito

Mock Creation

Option 1:

- MyObject o = Mockito.mock(MyObject.class)

Option 2:

- MyObject o1 = new MyObject();
- MyObject o2 = Mockito.spy(o1);

an NTT DATA Company

Mockito

Mockito.mock: Creates an object whose methods will give back always null. (Unless something different is configured aftwerward)

Mockito.spy: wraps the object passed in the creation. Unless something different is configured, the object will behave the same than the original one. Moreover, tracking functionality is added: what has been invoked, how many times….

Mockito

Stubbing a mock I

Returning an object:

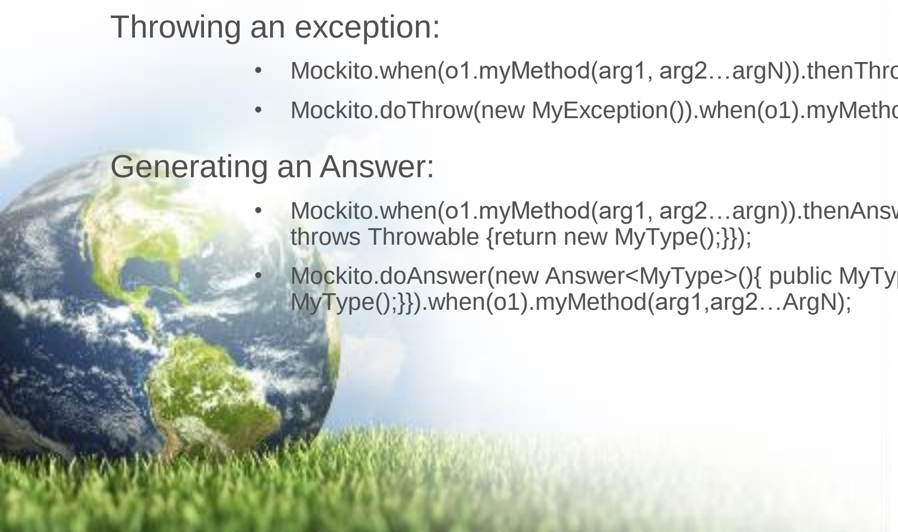
- Mockito.when(o1.myMethod(arg1, arg2…argN)).thenReturn(objectResult);
- Mockito.doReturn(objectResult).when(o1).myMethod(arg1, arg2…argN);
- Both are roughly equivalent but:
  - Option 1 cannot be used for stubbing void methods
  - Option 1 checks types in dev time
  - Option 1 calls the real method when using Mockito.spy during the configuration stage.

Throwing an exception:

- Mockito.when(o1.myMethod(arg1, arg2…argN)).thenThrow(new MyException());
- Mockito.doThrow(new MyException()).when(o1).myMethod(arg1,arg2,...argN);

Generating an Answer:

- Mockito.when(o1.myMethod(arg1, arg2…argn)).thenAnswer(new Answer<MyType>(){ public MyType answer(InvocationOnMock invocation) throws Throwable {return new MyType();}});
- Mockito.doAnswer(new Answer<MyType>(){ public MyType answer(InvocationOnMock invocation) throws Throwable {return new MyType();}}).when(o1).myMethod(arg1,arg2…ArgN);



Mockito

Stubbing a mock II

When Stubbing a mock it is possible to do it for a concrete or a generic input:

- Generic: Mockito.when(o1.metodo(Matchers.any(MyObject.class), Matchers.anyLong()…)).
- Concrete: Mockito.when(o1.metodo(Matchers.eq(o2), Matchers.eq(1l)…);

It is not allowed to mix both approaches:

- Mockito.when(o1.metodo(Matchers.eq(o2), 1)…); this will raise the next Exception: org.mockito.exceptions.misusing.InvalidUseOfMatchersException: Invalid use of argument matchers!

If “eq” matcher is used, it must be done on an object that implements “equals” method since this one will be invoke. NEVER use Object.equals(…) method unless you want to compare object instances themself!!

Mockito

Stubbing a mock III

- When stubbing a behaviour it is necessary to be careful with arguments matching:
  - Mockito.when(o1.metodo(Matchers.anyLong())).thenReturn(1l);
  - Mockito.when(o1.metodo(Matchers.anyLong())).thenReturn(2l);
- The last stubbing is the one that remains, hence, the answer will be the long value 2.

Mockito

Stubbing a mock IV

Some times stubbing a void method could be required.

To do this you need to create an Answer object for Void type: Mockito.doAnswer(new Answer<Void>(){ public Void answer(InvocationOnMock invocation) throws Throwable {

```
            //desired behaviour
            return null;}}).when(o1).myMethod(arg1, arg2,…argN);
```

doAnswer option is the only one allowed for Void types

Mockito

- Mockito allowes declarations with annotations:
- There are two ways to activate them:
  - @RunWith(MockitoJUnitRunner.class)
  - MockitoAnnotations.initMocks(this) inside @Before or @BeforeClass methods.
  - It is recommendable to use MockitoJUnitRunner since gives you automatic validation of framework usage, as well as an automatic initMocks().

Mockito

Annotations to init a Mock

@Mock: creates a mock and inject it in the attribute

@Spy: makes a wrap of an object to add tracking functionality. It requires that the attribute is initialized with new.

@InjectMocks: Instances the object an then injects mocks defined with @Mock y @Spy

Mockito

Assert

- Mockito adds verifying functionality for the Result Analysis step
- It works similar to Asserts
- Mockito.verify must be done on a mock created with Mockito.spy or @Spy.
- How to invoke it:
  - Mockito.verify(myObject).myMethod(…); will fail if myObject.myMethod was not invoked
  - Mockito.verify(myObject, Mockito.times(X)).myMethod(…); will fail if myObject.myMethod was not invoked X times
  - Second arguments is a class wich implements org.mockito.verification.VerificationMode
  - Mockito has some of them by default: atLeast, atMost…

04

Power Mockito

PowerMockito

- What if you need to mock a static method or a constructor?
- How to mock a private method?
- Is it possible to combine Spring Test Runer and PowerMockito Runner?
- Enters PowerMockito

PowerMockito

- It’s compatible with Mockito
- Only two annotations are required:
    - @RunWith(PowerMockRunner.class)
    - @PrepareForTest(<StaticClassToTest>.class)// If it is required to make a test on a class with static methods
- With these two annotations the standard Mockito functionality is provided plus PowerMockito functionality.
- Pay attention to @PrepareForTest: This annotation makes the junit be related to one class which is correct from methodology point of view.

PowerMockito

Mocking static method

// mock all the static methods in a class called “MyStaticClass" MyStaticClass); // use Mockito to set up your expectation

Mockito.when(Static.firstStaticMethod(param)).thenReturn(value);

PowerMockito.verifyStatic(Mockito.times(2));//Verifying invocation of the static method. In this example, it should be called twice

PowerMockito

Mocking constructor method

- Requires @PrepareForTest(<ClassToBeTested>.class)
- Mocking constructor is done via method “whenNew”.
- Examples:

```
PowerMockito.whenNew(MyClass.class).withNoArguments().thenThrow(new IOException("error message"));

MyClass mc = PowerMockito.mock(MyClass.class);
PowerMockito.whenNew(MyClass.class).withArguments(str1, str2,obj).thenReturn(mc );
```

PowerMockito

Mocking constructor method

- Requires @PrepareForTest(<ClassToBeTested>.class)
- Mocking constructor is done via method “whenNew”.
- Examples:

```
PowerMockito.whenNew(MyClass.class).withNoArguments().thenThrow(new IOException("error message"));

MyClass mc = PowerMockito.mock(MyClass.class);
PowerMockito.whenNew(MyClass.class).withArguments(str1, str2,obj).thenReturn(mc );
```

05

Methodologies : TDD & BDD

Methodologies: TDD & BDD

- TDD: Test Driven Development
- BDD: Behaviour Driven Development
- Bound to the concept of Continous Integration

Methodologies: TDD & BDD

TDD I

- TDD's Life Cycle:
  - Chose a requirement: it must be one that provides knowledge about the problem and it's easy to implement.
  - Write the test: Write the Unit Test for the requirement.
  - Verify the test fails: if it doesn't, then the test is wrong or the requirement was already implemented.
  - Write the Implementation: Write the simplest code that makes ther requirement works (KISS Principle)
  - Run automatic test: verify that test set works right.
  - Remove duplicities: Remove duplicated code. Make a small change every time and then run tests until it works.
  - Update Requirements list: Remove the implemented requirement from the list and add another one detected during the development.

Methodologies: TDD & BDD

TDD II

- Advantages:
    - Avoid unnecesary code
    - Produces trustable code since it is born from tests
    - Can guide the design of an application
    - Programmer only need the interfaces, not the implementation.
    - Maintanable code: Any modification will fastly tested since test are already there

Methodologies: TDD & BDD

TDD III

- Disadvantages:
    - It requires that tests can be automatic, thus GUI are dismissed (though there are some partial solutions)
    - Distributed object: This problem can be eased with Mocks
    - Data Bases: though there are solutions to set up a data base in memory, the data base itself must be defined before start. On the other hand, the data base should be adaptable and this is not always allowed or it has a high cost in project time

Methodologies: TDD & BDD

BDD I

- Combine TDD principles with Domain Driven Design and Object Oriented Design
- The aim of BDD is to provide shared tools and processes to Developers and Users
- This tools have as input Business specifications written in a structured natural language focussed on the domain
- This way it is avoided the wide range of tests between high level requirements and low level requirements as happens in TDD

Methodologies: TDD & BDD

BDD II

- One behaviour follows the structure of an user story in Agile
- It uses language from the Domain structured in such way that it is easy to process:
  - Given a game 5 x 5
  - When I toggle the cell at (3, 2)
  - Then then grid should look like this
- Bold words are reserved words

an NTT DATA Company

BECA Java

¿QUESTIONS?