

IT Bootcamp

Testing,

Introducción a Tests Unitarios, JUnit,
Unit Test con y sin Mock

DigitalHouse>

Índice

1. [Introducción a Tests Unitarios](#)
2. [JUnit](#)
3. [Unit Test sin Mocks](#)
4. [Unit Test con Mocks](#)

1 | Introducción a Test Unitarios

Niveles de Testing

En el testing de software hay 4 niveles principales de pruebas:



Pruebas Unitarias

Test de componentes individuales.

Comprueba que cada unidad funcione correctamente por separado



Pruebas de Integración

Test de componentes integrados.

Prueba la interacción entre las distintas partes del sistema



Pruebas de Sistema

Test del sistema completo.

Prueban el sistema en su totalidad



Pruebas de Aceptación

Test del sistema final.

Son pruebas realizadas en un entorno similar al de producción para demostrar que el sistema cumple las especificaciones y requisitos del cliente

Test Unitario

- Un test unitario es la unidad más pequeña de un sistema o aplicación que puede ser testeado, permite probar cada módulo por separado.
- Asegura que cada módulo del programa funcione para lo que fue diseñado, verifica que sean correctos los nombres y tipos de datos de los parámetros utilizados y lo que devuelve.
- El objetivo de estas pruebas es comprobar el correcto funcionamiento de los componentes de un programa de manera individual (por ejemplo el testing de una función o de una clase Java).
- Usualmente este tipo de testing es realizado por los desarrolladores, pero en la práctica puede ser realizada también por QAs (Quality Assurance).

¿Por qué realizamos Test Unitarios?

- Demuestran que la lógica del código funciona correctamente en diferentes escenarios.
- Permite identificar Bugs de manera temprana.
- Sirven como documentación del proyecto.
- Aumentan la legibilidad y ayudan a entender el código base.
- Reduce los costos.
- Se ejecutan en pocos segundos.
- Mejora el diseño y la calidad del código (TDD).

Actividad

Dada la siguiente clase Java identifica los comportamientos/escenarios que podrías testear unitariamente:

```
public class Calculator {  
    public int calculateTotal(int a, int b) {  
        return a + b;  
    }  
    public int calculateMinNumber(int a, int b) {  
        if ( a < b ) {  
            return a;  
        } else {  
            return b;  
        }  
    }  
}
```

*El método calculateTotal
puede ser testado
unitariamente*

*El método
calculateMinNumber puede
suponer dos test unitarios
ya que podemos
comprobar que **a** sea
menor que **b** y que
b sea menor que **a***

Las 3 A's del Unit Testing

Para llevar a cabo buenas pruebas unitarias, deben estar estructuradas siguiendo un proceso compuesto por tres pasos:

1

**Arrange
(Organizar)**

Primer paso de las pruebas unitarias.

Se definen los requisitos que debe cumplir el código.

2

**Act
(Actuar)**

**Paso intermedio de las pruebas,
llega el momento de hacer la llamada al código que queremos probar,
el cual dará lugar a los resultados a analizar.**

3

**Assert
(Afirmar)**

**Último paso,
es el momento de comprobar si los resultados obtenidos son los esperados.
Se valida que el componente a probar se comporte como lo habíamos previsto.**

Ejemplo

Función Principal

Test Unitarios

```
public class Add {  
    public int add (int a, int b) {  
        return a + b;  
    }  
}
```

```
@Test  
public void addTest () {  
    //Arrange  
    int number1 = 3; //input data  
    int number2 = 5; //input data  
    int expectedResult = 8; //expected output  
    Add addClass = new Add();  
    //Act  
    int result = addClass.add(number1 , number2);  
  
    //Assert  
    assertEquals(expectedResult, result);  
}
```

SUT - System Under Test

Cuando hacemos referencia a lo que estamos testeando lo llamamos Sistema Bajo Prueba o **SUT** por sus siglas en inglés (*System Under Test* o *Software Under Test*).

A red callout box with a white border and a small tail pointing towards the text "SUT".

```
public class StatusServiceTest {  
    @Test  
    public void testStatusOk() {  
        //Arrange  
        String expectedResult = "Status OK";  
        StatusService statusService = new StatusService();  
        //Act  
        String status = statusService .getStatus(1);  
        //Assert  
        Assert.assertEquals(expectedResult , status);  
    }  
}
```

```
public class StatusService {  
    public String getStatus(int statusCode) {  
        if (statusCode == 1) {  
            return "Status OK";  
        } else if (statusCode == 2) {  
            return "Status ERROR";  
        }  
    }  
}
```

El Principio F.I.R.S.T.

FIRST es el acrónimo de las cinco características que deben tener los **test unitarios** para ser considerados **de Calidad**.

Veamos cuales son esas 5 características:

FIRST es el acrónimo de las cinco características que deben tener los test unitarios para ser de Calidad

- **Fast (Rápido)**

Las pruebas unitarias deberían poder ejecutarse en pocos segundos, esto posibilita ejecutarlos de manera frecuente e identificar bugs de manera rápida y sencilla

- **Independent (Independiente)**

Cada prueba unitaria debe ser independiente de la otra, el resultado no debe verse alterado por el orden en que se ejecutan los test

- **Repeatable (Repetible)**

El resultado de las pruebas debe ser el mismo independientemente del entorno en que se ejecuten, no deben depender de la configuración de usuarios, hora de ejecución, etc.

- **Self-validating (Auto evaluable)**

El test debe ofrecer un resultado claro que indique si ha pasado o fallado, no debe requerir ninguna operación extra por parte del usuario

- **Timely (Oportuno)**

Las pruebas deben estar desarrolladas lo antes posible y siempre antes de subir el código de producción. Idealmente aplicar TDD

Cómo llevar a cabo Pruebas Unitarias

Este proceso puede realizarse de manera manual, aunque lo mejor es automatizar el procedimiento a través de herramientas.

Existen muchas opciones disponibles que varían en función del lenguaje de programación.

Estos son algunos ejemplos de **herramientas para desarrollar test unitarios en Java**:

The logo for JUnit, featuring a large green 'J' followed by 'Unit' in red.The logo for TestNG, with 'Test' in black, 'N' in red, and 'G' in yellow.The logo for Mockito, with the word 'mockito' in green and a small illustration of a glass with a straw and a leaf to the right.

Buenas prácticas



Las pruebas unitarias deben ser independientes.

Sigue un esquema claro.

Cualquier cambio realizado necesita pasar el test.

Corrige los bugs identificados durante las pruebas antes de continuar.

Realiza pruebas regularmente mientras programas.

Escribe test claros evitando la repetición de código y utilización innecesaria de recursos.

2 | JUnit



JUnit es uno de los frameworks de unit testing más populares en el ecosistema de Java.

La versión 5 de JUnit contiene innovaciones con respecto a su versión anterior, con el objetivo de soportar nuevas características de Java 8 y posteriores, así como permitir diferentes estilos de testing.



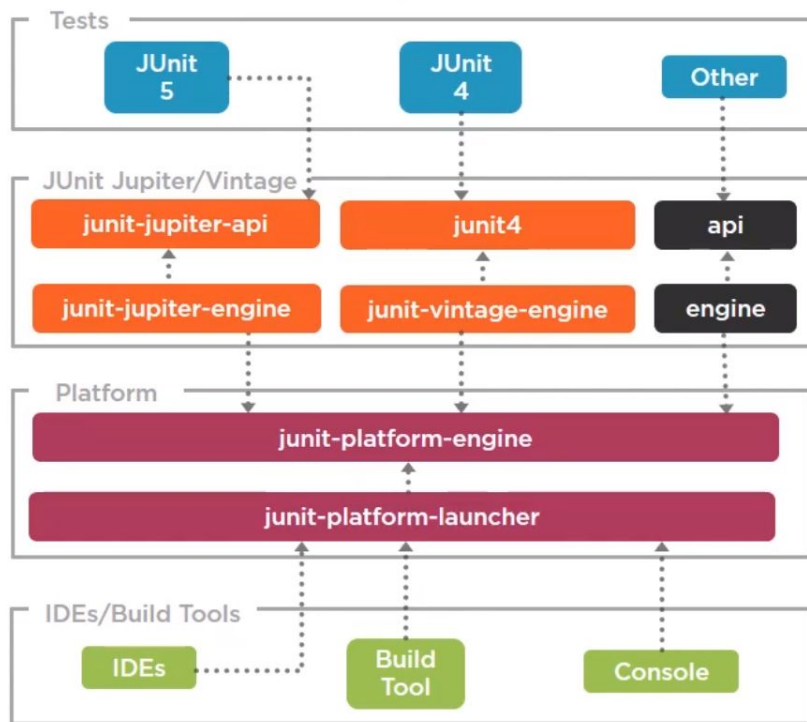
Arquitectura de JUnit 5

JUnit 5 se compone de diferentes módulos provenientes de 3 sub-proyectos diferentes:

JUnit Platform: Esta plataforma es responsable de ejecutar el framework de testing en la JVM. Define una interfaz estable y poderosa entre JUnit y sus clientes, es utilizada por las IDEs y las build tools. También define la API de TestEngine para desarrollar un framework de testing que corra en la plataforma de JUnit. Gracias a esto, se puede integrar librerías de testing de terceros, directamente en JUnit implementando custom TestEngine.

JUnit Jupiter: Este módulo incluye modelos nuevos de programación y extensiones para escribir tests en JUnit 5. Incluye nuevas anotaciones en comparación con JUnit 4.

JUnit Vintage: Soporta la ejecución de test en JUnit 3 y JUnit 4 basado en la plataforma de JUnit 5.



Spring Boot Starter Test

Para poder codificar nuestros test en JAVA podemos utilizar **spring-boot-starter-test**, el cual nos ofrece una serie de librerías que nos brindan las funcionalidades que necesitamos para realizar tareas de testing en nuestra aplicación.

Para utilizarlas debemos agregar la siguiente dependencia al pom.xml:

```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-test</artifactId>  
  <scope>test</scope>  
</dependency>
```

Spring Boot Starter Test

Al utilizar spring-boot-starter-test 'Starter' (en el scope de test), se podrá hacer uso de las siguientes librerías provistas:

- **JUnit** → Standard para unit testing en aplicaciones Java.
- **Spring Test & Spring Boot Test** → Utilidades y soporte para test de integración para aplicaciones Spring Boot.
- **AssertJ** → Librería de aserciones (Assertion library).
- **Hamcrest** → Librería de objetos matcher (también conocidos como constraints o predicates).
- **Mockito** → Framework para mocking en Java.
- **JSONassert** → Librería de aserciones para JSON.
- **JsonPath** → XPath para JSON.

Ejecutar Tests en IntelliJ

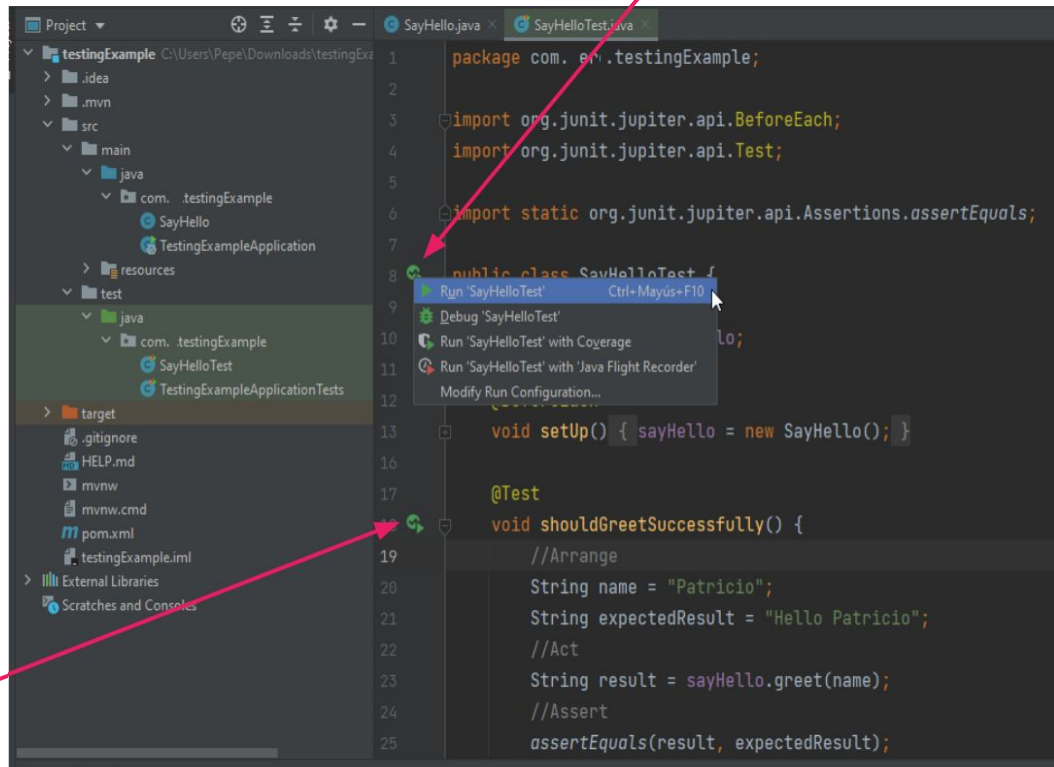
IntelliJ soporta JUnit 5 por defecto. Por lo tanto, ejecutar JUnit 5 en este IDE es bastante simple, simplemente se debe presionar

click derecho → Run sobre la clase de test o **Ctrl+Shift+F10**.

Dentro de la clase, al hacer click sobre el ícono de Test muestra las opciones de ejecución. Las más importantes son **"Run"** para una ejecución de corrido y **"Debug"** para una ejecución evaluativa.

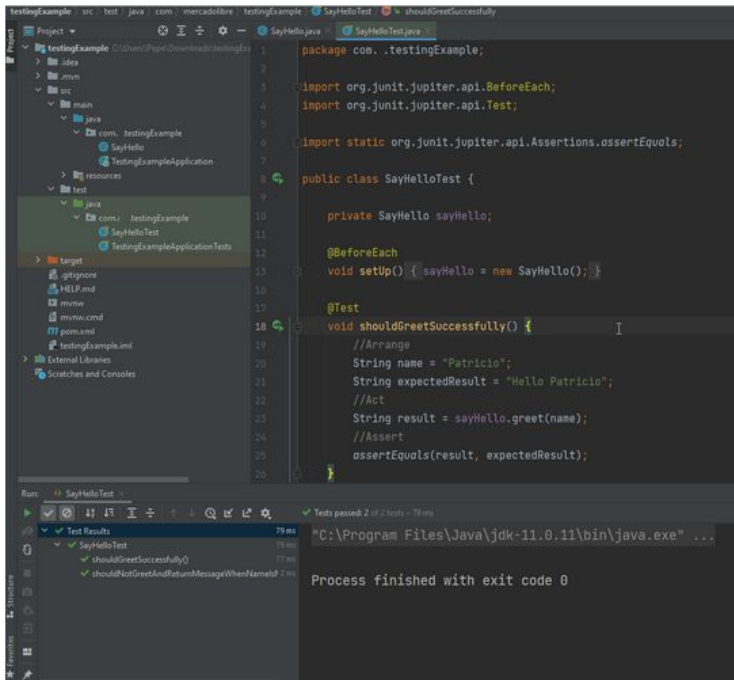
Si se utiliza el ícono del Método ejecutará solo el Test indicado

Si se utiliza el ícono de la Clase se ejecutarán todos los Tests



Consola de ejecución de test

Nos muestra el resultado de la ejecución de los tests, indicando con un **tilde verde** los que pasaron y con una **cruz amarilla** los que fallaron. Además nos informa la causa en el caso de los que no pasaron y permitiéndonos reiterar la ejecución.



The screenshot shows an IDE with a project named 'testingExample'. The left sidebar shows the project structure with folders for 'src', 'test', and 'target'. The main editor displays the code for 'SayHelloTest.java'. The code includes imports for JUnit and AssertJ, and defines a test class with a 'setUp' method and a 'shouldGreetSuccessfully' test method. The 'Run' button is highlighted. Below the code, the 'Run' output shows 'Tests passed: 2 of 2 tests - 79ms' and 'Process finished with exit code 0'.

```
package com.testingExample;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

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

public class SayHelloTest {

    private SayHello sayHello;

    @BeforeEach
    void setUp() { sayHello = new SayHello(); }

    @Test
    void shouldGreetSuccessfully() {
        //Arrange
        String name = "Patricio";
        String expectedResult = "Hello Patricio";
        //Act
        String result = sayHello.greet(name);
        //Assert
        assertEquals(result, expectedResult);
    }
}
```

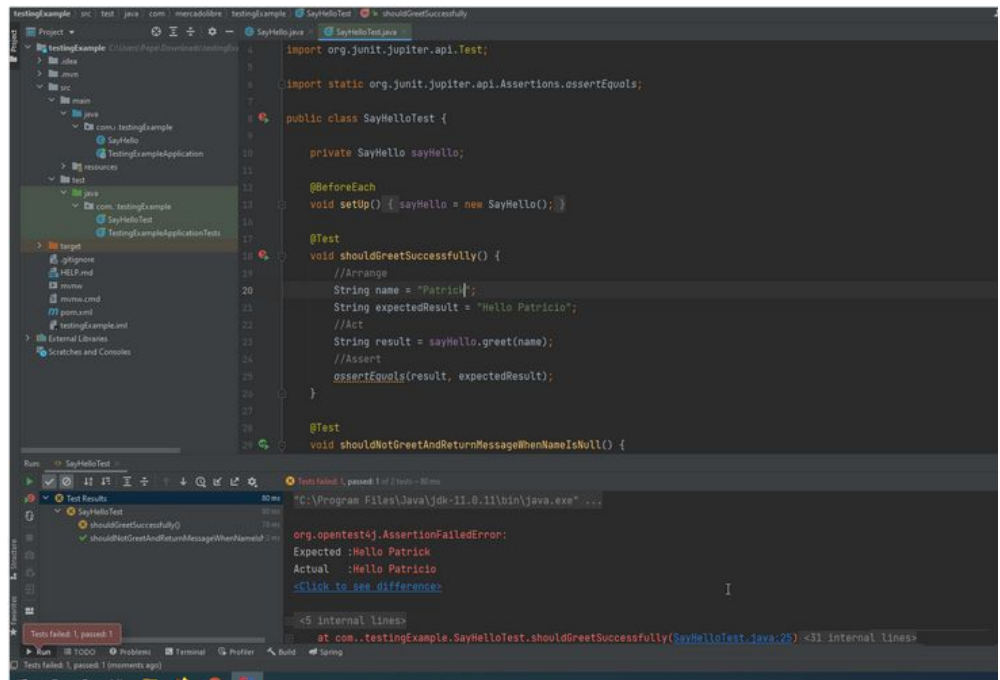
Run: SayHelloTest

Tests passed: 2 of 2 tests - 79ms

Test Results

- ✓ SayHelloTest
- ✓ shouldGreetSuccessfully() 79ms
- ✓ shouldNotGreetAndReturnMessageWhenNameIsNullOrEmpty() 77ms

Process finished with exit code 0



The screenshot shows the same IDE with the 'SayHelloTest.java' file. The 'Run' button is highlighted. Below the code, the 'Run' output shows 'Tests failed: 1, passed: 1 of 2 tests - 80ms'. The 'Test Results' section shows a failure for 'shouldGreetSuccessfully()'. The error message is 'org.opentest4j.AssertionFailedError: Expected :Hello Patrick Actual :Hello Patricio'. The 'Click to see difference' link is highlighted.

```
import org.junit.jupiter.api.Test;

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

public class SayHelloTest {

    private SayHello sayHello;

    @BeforeEach
    void setUp() { sayHello = new SayHello(); }

    @Test
    void shouldGreetSuccessfully() {
        //Arrange
        String name = "Patricio";
        String expectedResult = "Hello Patricio";
        //Act
        String result = sayHello.greet(name);
        //Assert
        assertEquals(result, expectedResult);
    }

    @Test
    void shouldNotGreetAndReturnMessageWhenNameIsNullOrEmpty() {
    }
}
```

Run: SayHelloTest

Tests failed: 1, passed: 1 of 2 tests - 80ms

Test Results

- ✗ SayHelloTest
- ✗ shouldGreetSuccessfully() 80ms
- ✓ shouldNotGreetAndReturnMessageWhenNameIsNullOrEmpty() 79ms

org.opentest4j.AssertionFailedError:
Expected :Hello Patrick
Actual :Hello Patricio
[Click to see difference](#)

<5 Internal lines
at com.testingExample.SayHelloTest.shouldGreetSuccessfully(SayHelloTest.java:25) <31 Internal lines

3 | Unit Test sin Mocks

DEMO:

Primer Test Unitario con
JUnit 5



Primer Test Unitario con JUnit 5

```
public class Calculator {  
    public int calculateTotal(int a, int b) {  
        return a + b;  
    }  
    public int calculateMinNumber(int a , int b) {  
        if ( a < b) {  
            return a;  
        } else {  
            return b;  
        }  
    }  
}
```

Para esto vamos a basarnos en la clase **Calculator** propuesta anteriormente, donde identificamos algunos escenarios que podemos testear

Anotaciones en JUnit

@Test → Indica que el método es un método de test.

@ParameterizedTest → Indica que se trata de un método parametrizado, es decir, permite correr el test con múltiples argumentos.

@DisplayName → Declara un nombre a mostrar customizado para la clase de test o método de test.

@Tag → Se utiliza para declarar etiquetas para filtrar tests, ya sea a nivel de clase o de método; análogo a los test groups en TestNG o categorías en JUnit 4.

@Disabled → Se utiliza para deshabilitar una clase de test o método de test, análogo a *@Ignore* de JUnit 4.

Anotaciones del ciclo de vida:

@BeforeEach → Ejecuta un método antes de cada *@Test*, *@RepeatedTest*, *@ParameterizedTest*, o al método *@TestFactory* en la clase actual; análogo a *@Before* de JUnit 4.

@AfterEach → Ejecuta un método después de cada *@Test*, *@RepeatedTest*, *@ParameterizedTest*, o *@TestFactory* método en la clase actual; análogo a *@After* de JUnit 4.

@BeforeAll → Ejecuta un método antes de todos los *@Test*, *@RepeatedTest*, *@ParameterizedTest*, y *@TestFactory* de la clase; análogo a *@BeforeClass* de JUnit 4.

@AfterAll → Ejecuta un método después de todos los *@Test*, *@RepeatedTest*, *@ParameterizedTest*, y *@TestFactory* de la clase; análogo a *@AfterClass* de JUnit 4.

Material extra



Puedes encontrar más información en los siguientes links

<https://www.sourcecodeexamples.net/2021/03/java-bean-validation-annotation-list.html>

<https://javaee.github.io/javaee-spec/javadocs/javax/validation/constraints/package-summary.html>

Aserciones en JUnit

JUnit 5 mantuvo muchos de los métodos de aserciones de JUnit 4 y añadió algunos nuevos que toman ventaja del soporte de Java 8. Adicionalmente, en esta versión de la librería, las aserciones se encuentran disponibles para todos los tipos primitivos, Objects, y Arrays.

- *assertArrayEquals*
- *assertEquals*
- *assertTrue* and *assertFalse*
- *assertNull* and *assertNotNull*
- *assertSame* and *assertNotSame*
- *assertAll*
- *assertNotEquals*
- *assertIterableEquals*
- *assertThrows*
- *assertTimeout* and *assertTimeoutPreemptively*
- *assertLinesMatch*

```
assertEquals(4, 4);
assertNotEquals(3, 4);
assertTrue(true);
assertFalse(false);
assertNull(null);
assertNotNull("Hello");
assertNotSame(originalObject, otherObject);
```

4 | **Unit Test con Mocks**

Double or Fake (Doble o Impostor)

Test Double es un término genérico para cualquier caso donde se deba reemplazar un objeto de producción con propósitos de testing.

Tipos de dobles o impostores:

- **Dummy** → Son objetos que se utilizan para realizar llamadas a otros métodos pero no se usan. Usualmente son utilizados para llenar listas de parámetros.
- **Stubs** → Proveen respuestas enlatadas a llamadas realizadas durante un test, se reprograman sus valores de retorno, los cuales serán constantes.
- **Spy** → Sólo un subconjunto de métodos son fake. A menos que explícitamente sean mockeados, el resto de los métodos son reales.
- **Mocks** → Están pre-programados con expectativas que forman una especificación de las llamadas que esperan recibir. Después de terminada la prueba puede ser examinado para comprobar si las interacciones con el SUT han sido correctas, se puede chequear si un método ha sido llamado y cuántas veces, si así lo fue. Al igual que los stub su comportamiento se puede pre-programar.

¿Por qué usar Mocks?

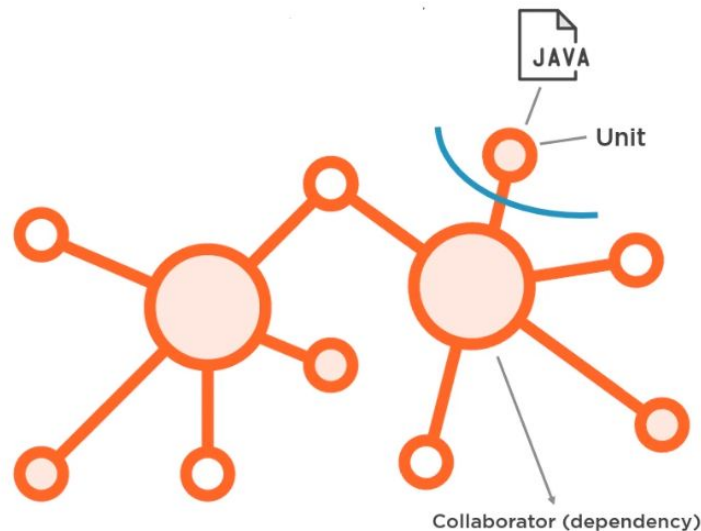
Permiten testear una **unidad de código aislada**, sin tener que preocuparse por sus dependencias.

Es posible aislar a la clase de sus colaboradores, **reemplazando las dependencias por mocks**, y, de esta manera testear todas las funcionalidades de esa unidad.

También es útil mockear una dependencia que aún no fue creada y se encuentra en proceso de desarrollo.

¿Qué colaboradores suelen mockearse?

Repositorios, Servicios, Librerías externas.



Mockito

Es el framework de mocks más conocido del mundo java. Permite crear y configurar objetos mock.

Otro framework bastante utilizado es [PowerMock](#).

Ofrece la posibilidad de mockear métodos estáticos, entre otras funcionalidades.



+ info

Para más información se puede visitar la documentación oficial en el siguiente enlace [Mockito.org](https://mockito.org)

Escribir un Test Unitario con Mocks

Anotar la clase con `@ExtendWith(MockitoExtension.class)` para inicializar los Mocks e inyectarlos en donde se indique.

`@Mock` permite mockear las **dependencias** de la clase.

Inyectar la dependencia mockeada mediante `@InjectMocks`

Dentro del test definir el comportamiento del mock:
when(methodCall).**thenReturn**(result)

Ejecutar el método de la clase a testear

Verificar que el método haya sido llamado y retorne los valores que esperábamos

```
@ExtendWith(MockitoExtension.class)
public class UserServiceTest {

    @Mock
    private UserRepository userRepo;

    @InjectMocks
    private UserService userService;

    @Test
    void testFindAllUsers() {
        List<UserDTO> expectedUsers = createUsersList();

        when(userRepo.findAll()).thenReturn(expectedUsers);

        List<UserDTO> currentUsers = userService.getAllUsers();

        verify(userRepo, atLeast(1)).findAll();
        assertEquals(expectedUsers, currentUsers);
    }
}
```


DigitalHouse>