Mockito

Mockito es un **framework de simulación (*mocking*) en Java**, ampliamente usado en pruebas unitarias.. Permite crear objetos simulados, especificar su comportamiento y verificar las interacciones con ellos, lo que facilita el aislamiento de las unidades de código bajo prueba.

Instalación de Mockito

Si usas Maven, agrega la siguiente dependencia en el archivo `pom.xml`. Maven se encargará de descargar y gestionar la librería en tu proyecto:

```
...
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>5.4.0</version>
    <scope>test</scope>
</dependency>
...
```

*Si es necesario, ajusta la versión según tus requerimientos.

Cómo se usa Mockito

- **Creación de objetos simulados (mocks):** En Mockito, puedes crear objetos simulados con el método mock(). Estos objetos actúan como sustitutos de los reales y pueden ser interfaces, clases concretas o abstractas.

- **Configuración de comportamientos:** Una vez que tienes un objeto simulado, puedes especificar su comportamiento utilizando métodos como when().thenReturn() o doAnswer(). Esto te permite definir qué debe hacer el objeto simulado cuando se llame a un método específico.

```
//Ejemplo: Creación de un mock y configuración de comportamiento

@Test
public void testMockitoMockCreationAndBehaviour() {
    // Creación del mock
    List<String> mockedList = mock(List.class);

    // Configuración del comportamiento
    when(mockedList.get(0)).thenReturn("primer elemento");

    // Prueba
    String firstElement = mockedList.get(0);
    assertEquals("primer elemento", firstElement);

  //Si el método no está configurado, Mockito devuelve null para objetos y valores por defecto para tipos primitivos (0 para int, false para boolean, etc.)
    String secondElement = mockedList.get(1);
    assertNull(secondElement);
}
```

- **Verificación de interacciones:** Con Mockito, puedes comprobar si se han realizado ciertas interacciones con los mocks. Se utiliza el método verify(), que permite verificar:

✔️ Si un método fue llamado

✔️ Cuántas veces fue llamado

✔️ Con qué argumentos fue llamado

```
@Test
public void testMockitoInteractionVerification() {
    // Creación del mock
    List<String> mockedList = mock(List.class);

    // Interacción con el mock
    mockedList.add("un elemento");

    // Verificación de la interacción
    verify(mockedList).add("un elemento");
```

```
}
```

- **Captura de argumentos (ArgumentCaptor):** Mockito permite capturar los argumentos pasados a los métodos simulados. Esto es útil cuando necesitas verificar los valores utilizados en una interacción.

```
@Test
public void testMockitoArgumentCaptor() {
    // Creación del mock
    List<String> mockedList = mock(List.class);

    // Interacción con el mock
    mockedList.add("un elemento");

    // Captura de argumentos
                        ArgumentCaptor<String>      argCaptor      =
ArgumentCaptor.forClass(String.class);
    verify(mockedList).add(argCaptor.capture());

    // Prueba
    assertEquals("un elemento", argCaptor.getValue());
}
```

- **Creación de objetos espías (Spies):** Un spy en Mockito permite rastrear y modificar el comportamiento de un objeto real durante las pruebas. A diferencia de los mocks, los spies mantienen la implementación original y solo sobrescriben los métodos necesarios.

```
@Test
public void testMockitoSpy() {
    // Creación del espía a partir de un ArrayList real
    List<String> spyList = spy(new ArrayList<>());

    // Interacción con el espía
    spyList.add("elemento real");

    // Verificación de la interacción
    verify(spyList).add("elemento real");

    // Prueba
    assertEquals(1, spyList.size());
    assertEquals("elemento real", spyList.get(0));
}
```

Diferencia entre mock y spy:

- En un **mock**, si no se configura un método, este devuelve el valor por defecto de su tipo de retorno.
- En un **spy**, los métodos mantienen su comportamiento original, salvo que se sobrescriban explícitamente.

- **Simulación de dependencias:** Uno de los principales beneficios de Mockito es que permite simular dependencias externas, lo que facilita el aislamiento de la unidad bajo prueba.

Cuando una clase depende de otra, en las pruebas unitarias se recomienda reemplazar la dependencia con un *mock* para evitar efectos secundarios y facilitar la validación del comportamiento. Esto permite probar la lógica interna de la clase sin necesidad de depender de servicios externos o implementaciones concretas.

```
//Ejemplo: Simulación de una dependencia en una clase
public class OrderProcessor {
    private PaymentGateway paymentGateway;

    public OrderProcessor(PaymentGateway paymentGateway) {
        this.paymentGateway = paymentGateway;
    }

    public boolean processOrder(double amount) {
        return paymentGateway.processPayment(amount);
    }
}
```

```
//Ejemplo: Prueba con mock para simular la dependencia

public class OrderProcessorTest {

    @Test
    public void testOrderProcessor() {
        // Crear un objeto simulado de PaymentGateway
                                PaymentGateway    mockPaymentGateway    =
mock(PaymentGateway.class);

        // Configurar el comportamiento del objeto simulado

when(mockPaymentGateway.processPayment(100.0)).thenReturn(true);
```

```
        // Inyectar la dependencia simulada en OrderProcessor
                            OrderProcessor    orderProcessor    =    new
OrderProcessor(mockPaymentGateway);

        // Probar la funcionalidad de OrderProcessor
        boolean result = orderProcessor.processOrder(100.0);

        // Verificar la interacción y resultado
        verify(mockPaymentGateway).processPayment(100.0);
        assertTrue(result);
    }
}
```

Aquí, **OrderProcessor** depende de **PaymentGateway**, pero en la prueba reemplazamos la implementación real con un **mock**, lo que nos permite **aislar la clase bajo prueba**.

✅ **Ventajas de usar mocks para dependencias**:

- Permite aislar pruebas unitarias
- Permite **probar lógica interna sin depender de servicios externos**
- Hace que las pruebas sean **más rápidas y confiables**

Mockito es una herramienta poderosa para la creación de pruebas unitarias en Java. Con su uso, puedes:

- Simular objetos y definir su comportamiento
- Verificar interacciones y capturar argumentos
- Usar *spies* para modificar objetos reales
- Simular dependencias para pruebas más confiables

💡 La inyección de dependencias es un concepto clave en diseño de software. Generalmente, se utiliza en el contexto del patrón experto. Si deseas profundizar en este tema, te dejamos este recurso: 👉 **Patrón Experto**.