Mockito

… EN DESARROLLO …

Introducción

Mockito es una popular biblioteca de Java diseñada específicamente para la creación de objetos simulados, conocidos como ***"mocks"***, en el contexto de pruebas unitarias. La principal ventaja de usar *"mocks"* es la capacidad de **simular el comportamiento de objetos complejos y sus dependencias**, permitiendo a los desarrolladores centrarse en probar la funcionalidad específica de la clase bajo prueba sin preocuparse por el estado o el comportamiento de sus colaboradores.

Mockito se ha convertido en una herramienta indispensable para los desarrolladores de Java debido a su facilidad de uso y su capacidad para mejorar significativamente la eficacia y la precisión de las pruebas unitarias. A través de **Mockito**, los desarrolladores pueden crear simulaciones controladas de objetos y definir cómo estos objetos deben comportarse durante la ejecución de las pruebas. Esto es especialmente útil en escenarios donde las dependencias externas, como bases de datos, servicios web o componentes de infraestructura, pueden introducir variabilidad y complejidad en las pruebas.

El uso de **Mockito** proporciona varios beneficios clave:

- **Aislamiento y Control**: Mockito permite aislar la clase bajo prueba de sus dependencias. Esto es esencial para asegurar que las pruebas unitarias sean precisas y se enfoquen únicamente en la funcionalidad de la clase bajo prueba. Al controlar el comportamiento de los *"mocks"*, los desarrolladores pueden simular diferentes escenarios, incluyendo casos extremos y condiciones de error, para verificar cómo responde la clase bajo prueba.

- **Simplificación de Pruebas**: En aplicaciones complejas, configurar un entorno de prueba que incluya todas las dependencias reales puede ser una tarea ardua y propensa a errores. Mockito simplifica este proceso al permitir la creación de objetos simulados que replican el comportamiento de las dependencias reales, pero sin la necesidad de configuraciones complicadas.

- **Verificación de Interacciones**: Más allá de simplemente simular comportamientos, Mockito también ofrece la capacidad de verificar que ciertas interacciones ocurrieron como se esperaba. Esto incluye la verificación de llamadas a métodos específicos, con ciertos parámetros, en los *"mocks"*. Esta característica es fundamental para asegurar que la clase bajo prueba interactúa correctamente con sus dependencias.

- **Mejora de la Calidad del Código**: Al facilitar la creación de pruebas unitarias efectivas y específicas, Mockito contribuye a la mejora de la calidad del código. Las pruebas unitarias bien escritas ayudan a detectar y corregir errores en etapas tempranas del desarrollo, lo que reduce significativamente los costos asociados con la corrección de defectos en etapas posteriores del ciclo de vida del software.

- **Facilidad de Uso**: La sintaxis intuitiva y las potentes capacidades de Mockito lo convierten en una herramienta accesible tanto para desarrolladores novatos como para expertos. La comunidad activa y la abundancia de recursos y documentación también hacen que sea fácil encontrar ayuda y ejemplos para resolver problemas específicos.

En resumen, Mockito es una herramienta esencial en el arsenal de pruebas unitarias de cualquier desarrollador Java. Su capacidad para crear y gestionar *"mocks"*, junto con su facilidad de uso y la profundidad de sus funcionalidades, la convierten en una opción preferida para garantizar que las clases se comporten correctamente en aislamiento, contribuyendo así a la creación de software de alta calidad y robustez.

En las siguientes secciones, exploraremos en detalle cómo configurar Mockito, crear y utilizar *"mocks"*, definir comportamientos, verificar interacciones y aplicar las mejores prácticas para maximizar el valor de nuestras pruebas unitarias.

> Sección generada por ChatGPT

Configuración y dependencias

Para comenzar a usar Mockito en un proyecto de Java, se necesita añadir las dependencias adecuadas en el archivo de configuración.

Dependencias para Maven

Si el proyecto utiliza Maven, se añaden las dependencias de **Mockito** y **JUnit 5** en el archivo `pom.xml` de la siguiente manera:

```
<dependencies>
    <!-- JUnit 5 -->
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter</artifactId>
        <version>5.10.3</version>
        <scope>test</scope>
    </dependency>

    <!-- Mockito -->
    <dependency>
        <groupId>org.mockito</groupId>
        <artifactId>mockito-core</artifactId>
        <version>5.11.0</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.mockito</groupId>
        <artifactId>mockito-junit-jupiter</artifactId>
        <version>5.11.0</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Para simplificar la gestión de versiones y asegurar la compatibilidad entre las diferentes partes de Mockito, se puede utilizar el **BOM (Bill of Materials)** de Mockito y JUnit.

Se añade el BOM en la sección `<dependencyManagement>` del archivo `pom.xml`:

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.junit</groupId>
            <artifactId>junit-bom</artifactId>
            <version>5.10.3</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
        <dependency>
            <groupId>org.mockito</groupId>
            <artifactId>mockito-bom</artifactId>
            <version>5.11.0</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>

<dependencies>
    <!-- JUnit 5 -->
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter</artifactId>
        <scope>test</scope>
```

```
    </dependency>

    <!-- Mockito -->
    <dependency>
        <groupId>org.mockito</groupId>
        <artifactId>mockito-core</artifactId>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.mockito</groupId>
        <artifactId>mockito-junit-jupiter</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Usar el BOM garantiza que todas las dependencias relacionadas con Mockito y JUnit usen **versiones compatibles entre sí**.

Dependencias para Gradle

Si el proyecto utiliza Gradle, añadir las dependencias de **Mockito** y **JUnit 5** en el archivo `build.gradle`:

```
dependencies {
    // JUnit 5
    testImplementation('org.junit.jupiter:junit-jupiter:5.10.3')

    // Mockito
    testImplementation('org.mockito:mockito-core:5.11.0')
    testImplementation('org.mockito:mockito-junit-jupiter:5.12.0')
}
```

En Gradle, también se puede utilizar el **BOM (Bill of Materials)** de Mockito y JUnit para manejar las versiones de manera más sencilla.

Se añade el BOM en la configuración del proyecto:

```
dependencies {
    // Importar el BOM de Mockito y JUnit
    testImplementation(platform('org.junit:junit-bom:5.10.3'))
    testImplementation(platform('org.mockito:mockito-bom:5.12.0'))

    // JUnit 5
    testImplementation('org.junit.jupiter:junit-jupiter')

    // Mockito
    testImplementation('org.mockito:mockito-core')
    testImplementation('org.mockito:mockito-junit-jupiter')
}
```

Utilizando el BOM de Mockito y JUnit, no se necesita especificar las versiones individuales para `mockito-core` y `mockito-junit-jupiter`, ya que el BOM se encarga de establecer las versiones compatibles automáticamente.

Creación de Mocks

Una de las características principales de Mockito es la capacidad de crear objetos simulados, conocidos como *"mocks"*. Estos *"mocks"* permiten a los desarrolladores simular el comportamiento de objetos complejos y sus dependencias en un entorno controlado.

La forma más básica de crear un *"mock"* en Mockito es utilizando el método estático `Mockito.mock()`. Este método toma la clase del objeto que se quiere simular y devuelve una **instancia simulada de esa clase**.

```
// ENTIDAD que se quiere SIMULAR
public interface Service {
    String performOperation();
}
```

Se crea un *"mock"* de esta interfaz en un prueba unitaria de la siguiente manera:

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

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

public class ServiceTest {

    @Test
    public void testPerformOperation() {
        // Crear el mock de la interfaz Service
        Service mockService = Mockito.mock(Service.class);

        // Definir el comportamiento del mock
        Mockito.when(mockService.performOperation()).thenReturn("Mocked Response");

        // Utilizar el mock en una prueba
        String response = mockService.performOperation();

        // Verificar el resultado
        assertEquals("Mocked Response", response);
    }
}
```

Mockito también proporciona una manera más limpia y concisa de crear *"mock"* utilizando **anotaciones**. Para esto, se usa la anotación `@Mock`.

- Mockito's Mock Methods - Baeldung

Definición de Comportamiento

Además de crear los *"mocks"*, hay que **definir cómo deben comportarse** cuando se les llamen ciertos métodos. Esto se hace utilizando los métodos `when().thenReturn()` y `when().thenThrow()` de Mockito.

Sin embargo es importante mencionar una consideración clave: definir el comportamiento de los *"mocks"* puede llevar a pruebas que **no reflejen fielmente el comportamiento real** de las clases que se están simulando. Esto puede ocurrir si el comportamiento definido en los *"mocks"* no coincide con la implementación real de las clases.

Por lo tanto, es crucial utilizar esta técnica con cuidado y sólo definir un comportamiento con aquellas entidades que realmente deben ser simuladas en las pruebas como por ejemplo una llamada a una API externa o a una base de datos.

El método `when().thenReturn()` se utiliza para especificar el valor de retorno de un método de un *"mock"* que simula una clase cuando este método es invocado.

```
// ENTIDAD que se quiere SIMULAR
public interface Service {
    String performOperation();
```

```
    String performOperationWithArgs(String arg);
}
```

En el ejemplo, se define que cuando se invoque el método `mockService.performOperation()` se devolverá la cadena "Mocked Response":

```
@Test
public void testPerformOperation() {
    // Crear el mock de la interfaz Service
    Service mockService = Mockito.mock(Service.class);

    // Definir el comportamiento del mock
    when(mockService.performOperation()).thenReturn("Mocked Response");

    // Utilizar el mock en una prueba
    String response = mockService.performOperation();

    // Verificar el resultado
    assertEquals("Mocked Response", response);
}
```

También se puede definir comportamientos de **métodos que aceptan parámetros**:

```
@Test
public void testPerformOperation() {
    // Crear el mock de la interfaz Service
    Service mockService = Mockito.mock(Service.class);

    // Definir el comportamiento del mock para diferentes parámetros
    when(mockService.performOperationWithArgs("input1")).thenReturn("Output1");
    when(mockService.performOperationWithArgs("input2")).thenReturn("Output2");

    // Utilizar y verificar el resultado
    assertEquals("Output1", mockService.performOperationWithArgs("input1"));
    assertEquals("Output2", mockService.performOperationWithArgs("input2"));
}
```

Se puede definir diferentes comportamientos para múltiples llamadas consecutivas al mismo método **encadenando llamadas** a `thenReturn()` seguido de más valores:

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

    @Mock
    private List<String> mockList;

    @Test
    public void testGetSizeConsecutiveCalls() {
        // Configurar el comportamiento del mock para devolver valores diferentes en llamadas consecutivas
        when(mockList.size()).thenReturn(1).thenReturn(2).thenReturn(3);

        // Añadir elementos a la lista
        mockList.add("Element1");
        mockList.add("Element2");

        // Verificar el tamaño en llamadas consecutivas
        assertEquals(1, mockList.size()); // Primera llamada
        assertEquals(2, mockList.size()); // Segunda llamada
        assertEquals(3, mockList.size()); // Tercera llamada
    }
}
```

El método `when().thenThrow()` se utiliza para especificar que un método de un *"mock"* debe lanzar una excepción cuando se le llama con ciertos parámetros:

```
@Test
public void testPerformOperation() {
    // Crear el mock de la interfaz Service
    Service mockService = Mockito.mock(Service.class);

    // Definir el comportamiento del mock
    when(mockService.performOperation()).thenThrow(new RuntimeException("Mocked Exception"));

    // Utilizar el mock en una prueba
    String response = mockService.performOperation();

    // Verificar el resultado
    assertEquals("Mocked Response", response);
}
```

En el ejemplo, se define que cuando se invoque el método `mockService.performOperation()`, el *"mock"* lanzará una `RuntimeException` con el mensaje "Mocked Exception".

Para **métodos void**, no puede utilizar `when().thenReturn()` ya que no devuelven ningún valor. En su lugar, puedes usar `doThrow()`, `doAnswer()`, `doNothing()`, etc.

- Mockito When/Then Cookbook - Baeldung
- The Difference Between doAnswer() and thenReturn() in Mockito - Baeldung

Verificación de Interacciones

Una de las capacidades más importantes de Mockito es la posibilidad de **verificar interacciones con los *mocks***. Esto permite asegurar de que los métodos en los *mocks* se llaman como se espera durante la ejecución del código.

El método `verify()` se utiliza para **verificar la invocación** de un método en un *mock*:

```
@Test
public void testPerformOperation() {
    // Crear el mock de la interfaz Service
    Service mockService = Mockito.mock(Service.class);

    // Llamar al método en el mock
    mockService.performOperation();

    // Verificar que el método se ha llamado exactamente una vez
    verify(mockService).performOperation();
}
```

Se puede especificar **el número exacto de veces** que se espera que un método sea llamado utilizando el método `times()`. También se puede usar `atLeast()`, y `atMost()` para especificar la cantidad de llamadas esperadas:

```
@Test
public void testPerformOperation() {
    // Crear el mock de la interfaz Service
    Service mockService = Mockito.mock(Service.class);

    // Llamar al método en el mock varias veces
    mockService.performOperation();
    mockService.performOperation();
```

```
    // Verificar que el método se ha llamado exactamente dos veces
    verify(mockService, times(2)).performOperation();
}
```

También se puede verificar que un método en un mock se ha llamado con **ciertos parámetros**:

```
@Test
public void testPerformOperation() {
    // Crear el mock de la interfaz Service
    Service mockService = Mockito.mock(Service.class);

    // Llamar al método en el mock con un parámetro específico
    mockService.performOperationWithArgs("testInput")

    // Verificar que el método se ha llamado con el parámetro "testInput"
    verify(mockService).performOperationWithArgs("testInput");
}
```

Se puede verificar que un método en un *mock* **nunca se ha llamado** utilizando el método `never()`.

```
@Test
public void testPerformOperation() {
    // Crear el mock de la interfaz Service
    Service mockService = Mockito.mock(Service.class);

    // No llamar al método en el mock

    // Verificar que el método nunca se ha llamado
    verify(mockService, never()).performOperation();
}
```

Para verificar que los métodos se han llamado en un orden específico, utilizamos la clase `InOrder`:

```
import static org.mockito.Mockito.*;

public class NotificationServiceTest {

    // ... setup y otros métodos

    @Test
    public void testNotifyUserCallsSendInOrder() {
        notificationService.notifyUser("Message 1");
        notificationService.notifyUser("Message 2");

        InOrder inOrder = inOrder(mockNotifier);

        // Verificar que send fue llamado primero con "Message 1" y luego con "Message 2"
        inOrder.verify(mockNotifier).send("Message 1");
        inOrder.verify(mockNotifier).send("Message 2");
    }
}
```

Para asegurarnos de que no se han hecho más interacciones con un *mock*, usamos `verifyNoMoreInteractions()`. También podemos usar `verifyZeroInteractions()` para asegurarnos de que no hubo ninguna interacción en absoluto.

```
import static org.mockito.Mockito.*;

public class NotificationServiceTest {
```

```
    // ... setup y otros métodos

    @Test
    public void testNoMoreInteractions() {
        notificationService.notifyUser("Message");

        // Verificar que no hubo más interacciones con el mock
        verifyNoMoreInteractions(mockNotifier);
    }

    @Test
    public void testZeroInteractions() {
        // Verificar que no hubo ninguna interacción con el mock
        verifyZeroInteractions(mockNotifier);
    }
}
```

- Mockito Verify Cookbook - Baeldung

Anotaciones en Mockito

Hay diferentes formas de **habilitar las anotaciones** con pruebas en Mockito.

La forma recomendable sería anotar la prueba JUnit con `@ExtendWith(MockitoExtension.class)`. En ese caso hay que importar la dependencia `mockito-junit-jupiter` con Maven o Gradle para poder hacer uso de `MockitoJUnitRunner`:

```
@ExtendWith(MockitoExtension.class)
public class MockitoAnnotationUnitTest {
    // ...
}
```

De forma alternativa, se puede habilitar las anotaciones de Mockito mediante programación invocando el método `MockitoAnnotations.openMocks()`:

```
@BeforeEach
public void init() {
    MockitoAnnotations.openMocks(this);
}
```

Finalmente, se puede utilizar `MockitoJUnit.rule()`:

```
public class MockitoAnnotationsInitWithMockitoJUnitRuleUnitTest {

    @Rule
    public MockitoRule initRule = MockitoJUnit.rule();

    ...
}
```

- Mockito and JUnit 5 – Using ExtendWith - Baeldung

@Mock

Una vez habilitadas las anotaciones, se pueden crear e inyectar *"mocks"* con la anotación `@Mock`, lo que facilita la creación y hace el código más limpio y compacto al no tener que invocar `Mockito.mock(...)` de forma manual:

```java
@ExtendWith(MockitoExtension.class)
public class ServiceTest {

    @Mock
    private Service mockService;

    @Test
    public void testPerformOperation() {
        // Definir el comportamiento del mock
        when(mockService.performOperation()).thenReturn("Mocked Response");

        // Utilizar el mock en una prueba
        String response = mockService.performOperation();

        // Verificar el resultado
        assertEquals("Mocked Response", response);
    }
}
```

@InjectMocks

Una característica poderosa de Mockito es la capacidad de inyectar *mocks* en la clase bajo prueba utilizando la anotación `@InjectMocks`. Esto simplifica la configuración de las pruebas al manejar automáticamente la creación y la inyección de los mocks en los campos de la clase que se está probando.

La anotación `@InjectMocks` se utiliza para crear una instancia de la clase bajo prueba y automáticamente inyectar los *mocks* en sus dependencias. Esta anotación facilita la configuración de las pruebas al evitar la necesidad de crear manualmente la instancia de la clase y configurar sus dependencias.

```java
// Modelo del dominio
public class User {
    private String name;

    public User(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

}
```

```java
// Repository
public interface UserRepository {
    User findUserByName(String name);
}
```

```java
// Servicio que utiliza 'UserRepository'
public class UserService {

    private UserRepository userRepository;

    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    public User getUserByName(String name) {
        return userRepository.findUserByName(name);
    }
```

```
}
```

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

    // Crear un mock de 'UserRepository'
    @Mock
    private UserRepository userRepository;

    // Crea un instancia de 'UserService' y se inyecta el mock de 'UserRepository'
    @InjectMocks
    private UserService userService;

    @Test
    public void testGetUserByName() {
        // Definir el comportamiento del mock
        User mockUser = new User("John Doe");
        when(userRepository.findUserByName("John Doe")).thenReturn(mockUser);

        // Llamar al método en la clase bajo prueba
        User user = userService.getUserByName("John Doe");

        // Verificar que el método en el mock se haya llamado correctamente
        verify(userRepository).findUserByName("John Doe");

        // Verificar el resultado
        assertEquals("John Doe", user.getName());
    }

}
```

Además de la inyección de dependencias en el constructor, `@InjectMocks` también puede inyectar dependencias en campos y métodos *'setter'*.

```
// Servicio
public class UserService {

    private UserRepository userRepository;

    // Constructor sin parámetros. No se realiza la inyección aquí
    public UserService() { }

    // Inyección en el método setter
    public void setUserRepository(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    public User getUserByName(String name) {
        return userRepository.findUserByName(name);
    }

}
```

@Captor

La anotación `@Captor` en Mockito se utiliza para capturar los argumentos pasados a los métodos de los *mocks*. Esto es útil cuando se quiere verificar no solo que un método se llamó, sino también que se llamó con los argumentos correctos.

La anotación `@Captor` declara y crea instancias de `ArgumentCaptor`. Un `ArgumentCaptor` permite capturar los argumentos que se pasan a los métodos de los *mocks*, lo que facilita la validación de estos argumentos en tus pruebas unitarias.

```java
// Repository
public interface UserRepository {
    void sendNotification(User user);
}
```

```java
// Servicio
public class UserService {
    private UserRepository userRepository;

    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    public void notifyUser(User user) {
        userRepository.sendNotification(user);
    }
}
```

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

    @Mock
    private UserRepository userRepository;

    @InjectMocks
    private UserService userService;

    @Captor
    private ArgumentCaptor<User> userCaptor;

    @Test
    public void testNotifyUser() {
        User user = new User("John Doe");

        // Llamar al método en la clase bajo prueba
        userService.notifyUser(user);

        // Capturar el argumento pasado al método sendNotification
        verify(userRepository).sendNotification(userCaptor.capture());

        // Obtener el valor capturado
        User capturedUser = userCaptor.getValue();

        // Verificar que el argumento capturado es el esperado
        assertEquals("John Doe", capturedUser.getName());
    }
}
```

La idea es poder "capturar" los argumentos que se están utilizando para hacer llamadas a métodos dentro de un método. Es decir, el método `userService.notifyUser(user);` es el método visible pero internamente desde ese tipo de métodos se pueden estar haciendo llamadas a otros métodos.

Con `ArgumentCaptor<?>` podemos "capturar" los argumentos utilizados para hacer esas llamadas. En el ejemplo, se "capturan" los argumentos usados en `userRepository.sendNotification(user);` que es el método interno. Sin `ArgumentCaptor<?>` no se podría comprobar como se están usando los argumentos ya que aunque se conoce que argumentos se le están pasando al método más visible, no se sabe que se hace con esos argumentos internamente.

@Spy

La anotación `@Spy` en Mockito se utiliza para crear un espía (*spy*) de un objeto real. A diferencia de los *mocks*, que son objetos simulados, los espías permiten espiar (o monitorear) las llamadas a un objeto real, al mismo tiempo que se puede configurar su comportamiento.

Un *spy* en Mockito es un objeto real en el que se pueden espiar los métodos. Permite observar las interacciones con el objeto y, opcionalmente, configurar respuestas personalizadas para ciertos métodos. A diferencia de los *mocks*, que no contienen ningún comportamiento real, los espías mantienen el comportamiento original del objeto, a menos que se especifique lo contrario.

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

    @Spy
    private List<String> spyList = new ArrayList<>();

    @Test
    public void testSpy() {
        spyList.add("Hello");

        // Verificar que se haya llamado al método add
        verify(spyList).add("Hello");

        // Verificar que el comportamiento real se mantenga
        assertEquals(1, spyList.size());
        assertEquals("Hello", spyList.get(0));
    }
}
```

- Getting Started with Mockito @Mock, @Spy, @Captor and @InjectMocks - Baeldung

Argument Matchers

`ArgumentMatchers` es una clase de utilidad en Mockito que proporciona **una serie de métodos estáticos** para hacer coincidir argumentos en métodos mockeados. Estos *matchers* permiten realizar verificaciones y configuraciones más dinámicas en lugar de utilizar valores exactos.

Mockito proporciona una variedad de matchers que puedes usar para diferentes propósitos:

- `any()` : Coincide con cualquier valor del tipo especificado.
    - `anyInt()` : Coincide con cualquier entero (`int`).
    - `anyDouble()` : Coincide con cualquier valor de punto flotante (`double`).
    - `anyFloat()` : Coincide con cualquier valor de punto flotante (`float`).
    - `anyLong()` : Coincide con cualquier valor largo (`long`).
    - `anyShort()` : Coincide con cualquier valor corto (`short`).
    - `anyByte()` : Coincide con cualquier valor de byte (`byte`).
    - `anyChar()` : Coincide con cualquier carácter (`char`).
    - `anyBoolean()` : Coincide con cualquier valor booleano (`boolean`).
    - `anyString()` : Coincide con cualquier cadena (`String`).
    - `anyList()` : Coincide con cualquier lista (`List`).
    - `anyMap()` : Coincide con cualquier mapa (`Map`).
- `eq()` : Coincide con un valor específico.
    - `eq(5)` : Coincide con el valor entero 5.
    - `eq("Test")` : Coincide con la cadena "Test".
- `isNull()` : Coincide con valores `null` .

- `isNotNull()` : Coincide con cualquier valor no `null` .
- `argThat()` : Permite utilizar una expresión lambda o un predicado para hacer coincidir argumentos que cumplen con una condición específica.
    - `argThat(value -> value.length() > 5)` : Coincide con argumentos que cumplen con la condición proporcionada.
- `contains()` : Coincide con cadenas que contienen un valor específico.
- `startsWith()` : Coincide con cadenas que comienzan con un valor específico.
- `endsWith()` : Coincide con cadenas que terminan con un valor específico.

En este ejemplo se usa `anyInt()` para indicar qué debe devolver el *mock* para cualquier entero como argumento:

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

    @Mock
    private MyService myService;

    @Test
    public void testServiceMethod() {
        // Configurar el mock para devolver un valor cuando se llama con cualquier entero
        when(myService.doSomething(anyInt())).thenReturn("Result");

        String result = myService.doSomething(123);

        assertEquals("Result", result);
    }
}
```

Los *matchers* también se utilizan para **verificar** que los métodos fueron llamados con ciertos argumentos:

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

    @Mock
    private MyService myService;

    @Test
    public void testVerifyMethodCall() {
        myService.doSomething(123);

        // Verificar que el método fue llamado con cualquier entero
        verify(myService).doSomething(anyInt());
    }
}
```

Además, se pueden crear ***matchers* personalizados** usando `argThat()` para especificar condiciones más complejas:

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

    @Mock
    private MyService myService;

    @Test
    public void testCustomMatcher() {
        when(myService.process(argThat(value -> value.length() > 5)))
            .thenReturn("Long String");
```

```
        String result = myService.process("HelloWorld");

        assertEquals("Long String", result);
    }
}
```

- Mockito ArgumentMatchers - Baeldung

Buenas Prácticas

Estas prácticas incluyen la gestión adecuada de mocks, la claridad en las pruebas, y cómo evitar problemas comunes.

Mantener las pruebas simples y claras

La claridad en las pruebas es esencial para mantener la calidad del código y facilitar la comprensión.

Cada prueba debe ser **independiente de las demás**. Hay que asegurarse de que la configuración de un test no afecte a otros tests.

Utilizar **nombres descriptivos** para los métodos de prueba que indiquen claramente qué aspecto del comportamiento se está probando.

Configura solo los *mocks* y *stubs* que sean **necesarios para la prueba**. Evitar configurar comportamientos innecesarios que no aporten a la verificación.

No abusar de los *mocks*

Utilizar mocks es útil, pero el abuso de ellos puede llevar a pruebas frágiles.

Evitar crear *mocks* para cada dependencia de una clase. En su lugar, considerar el uso de objetos reales o `@Spy` para dependencias complejas.

No configurar *mocks* para **comportamientos extremadamente complejos**. En su lugar, considerar simplificar el diseño del código.

Usar `@InjectMocks` con precaución

La anotación `@InjectMocks` es potente, pero puede hacer que las pruebas sean menos predecibles si no se usa correctamente.

Controlar la inyección asegurándose de que la clase que se está probando tenga un estado consistente después de la inyección de mocks.

Si una clase tiene **demasiadas dependencias**, considera refactorizar para hacer la clase más modular y fácil de probar.

Verificar las interacciones solo cuando sea necesario

La verificación de interacciones debe ser utilizada para validar que los métodos se llamaron como se esperaba. No es necesario verificar cada llamada; usar esta característica solo cuando **la interacción sea parte del comportamiento esperado**.

Refactorizar las pruebas regularmente

Al igual que el código de producción, **las pruebas también deben ser mantenidas y refactorizadas**. Revisar y actualizar las pruebas para asegurarse de que siguen siendo relevantes y efectivas a medida que el código evoluciona.

Referencias

- https://site.mockito.org/
- https://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/Mockito.html
- https://www.baeldung.com/mockito-series

Licencia