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` en combinación con **la extensión de JUnit 5**.

Para usar las **anotaciones** hay que configurar la clase de prueba para que use la entensión de Mockito con `@ExtendWith(MockitoExtension.class)`. Una vez configurada, se pueden crear *"mocks"* con la anotación `@Mock`, lo que facilita la creación de los *"mocks"* y hace el código más limpio:

```
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.mockito.Mockito.when;

@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);
    }
}
```

- Mockito and JUnit 5 – Using ExtendWith - 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"));
}
```

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".

- Mockito When/Then Cookbook - Baeldung

Verificación de Interacciones

TODO

Anotaciones de Mockito

TODO

Argument Matchers

TODO

Simulación de Excepciones

TODO

Ejemplos Completos

TODO

Resumen y Buenas Prácticas

TODO

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