UTN
Facultad Regional Córdoba

TUP

Tecnicatura Universitaria en Programación

LABORATORIO DE COMPUTACIÓN III

Unidad Temática N°2:
Testing

Material de Estudio
2° Año – 3° Cuatrimestre

Índice

TESTING

Introducción

El testing es una parte fundamental de la programación, especialmente en proyectos de gran envergadura. Se trata de **un proceso que consiste en evaluar el comportamiento de un sistema**, aplicación o módulo de software **con el objetivo de asegurarse de que funciona de manera adecuada y cumple con los requerimientos especificados**.

En el contexto de la programación en Java, existen varias herramientas y frameworks para llevar a cabo el testing de aplicaciones. Uno de los más populares es **JUnit**, una biblioteca de pruebas unitarias que permite crear y ejecutar pruebas automatizadas para comprobar el funcionamiento de clases, métodos y otros componentes de un programa.

Además de JUnit, existen otras herramientas como TestNG y Mockito, que ofrecen diferentes funcionalidades para el testing en Java. La idea es que, a través del testing, se puedan identificar errores, bugs o problemas de rendimiento antes de que el software sea entregado al usuario final, lo que permite mejorar la calidad y la eficiencia del proceso de desarrollo de software.

Fundamentos del Testing

Los fundamentos del testing son los conceptos básicos que se deben comprender para llevar a cabo el proceso de evaluación y verificación de un software. A continuación, se presentan algunos de los principales fundamentos del testing:

- **Objetivo del testing**: El objetivo del testing es **identificar errores, bugs o problemas en el software** para mejorar su calidad y garantizar que cumple con los requerimientos especificados.
- **Tipos de testing**: Existen **diferentes tipos de testing que se aplican en diferentes etapas del proceso de desarrollo de software**, como pruebas unitarias, de integración, de sistema, de aceptación, entre otras.
- **Características de una buena prueba**: Las pruebas deben **ser objetivas, reproducibles, relevantes, precisas y completas**, para que puedan ser utilizadas de manera efectiva para identificar errores.
- **Herramientas de testing**: Existen varias herramientas y frameworks de testing, como JUnit, TestNG, Selenium, Mockito, entre otros, que se utilizan para llevar a cabo diferentes tipos de pruebas.

- **Ciclo de vida del testing**: El testing **es un proceso continuo que se lleva a cabo a lo largo de todo el ciclo de vida del software**, desde la fase de diseño hasta la de mantenimiento.
- **Planificación y diseño de pruebas**: La planificación y el diseño de las pruebas es una parte fundamental del proceso de testing, ya que permite **identificar los requerimientos y las funcionalidades que deben ser probadas**, y diseñar casos de prueba que cubran todos los escenarios posibles.
- **Ejecución de pruebas**: La ejecución de las pruebas consiste en **ejecutar los casos de prueba diseñados y recopilar los resultados** para identificar errores o problemas.
- **Análisis y reporte de pruebas**: El análisis y reporte de las pruebas es una parte importante del proceso de mejora continua, ya que **permite identificar los errores y problemas encontrados y utilizar esta información para mejorar el software**.

Comprender estos fundamentos es esencial para llevar a cabo un proceso de testing efectivo y garantizar la calidad del software.

Tipos de Testing

Existen diferentes tipos de testing que se aplican en diferentes etapas del proceso de desarrollo de software. A continuación, te presento los principales tipos de testing, para qué se usan y en qué etapa del proceso se aplican:

- **Pruebas Unitarias**: Las pruebas unitarias **son el tipo de prueba más básico y se centran en comprobar que cada unidad de código** (clase, método o función) funciona correctamente de manera aislada. Se realizan en la etapa de desarrollo y se ejecutan con frecuencia durante la fase de pruebas de unidad.
- **Pruebas de Integración**: Las pruebas de integración son el siguiente nivel de prueba después de las pruebas unitarias. Se utilizan para **probar la interacción entre diferentes unidades de código y verificar que funcionan correctamente juntas**. Las pruebas de integración pueden ser horizontales (pruebas de integración de un conjunto de unidades en un mismo nivel de abstracción) o verticales (pruebas de integración entre

diferentes niveles de abstracción). Estas pruebas se realizan en la etapa de integración.

- **Pruebas de Sistema**: Las pruebas de sistema se centran en **verificar que el software cumple con los requerimientos del sistema** y que funciona correctamente **en el contexto del entorno de producción**. Se realizan en la etapa de pruebas de sistema y pueden incluir pruebas de funcionalidad, pruebas de rendimiento, pruebas de carga y pruebas de seguridad.
- **Pruebas de Aceptación**: Las pruebas de aceptación son **realizadas por el cliente o usuario final para verificar que el software cumple con los requerimientos especificados** y es capaz de satisfacer las necesidades del usuario. Estas pruebas se realizan en la etapa de pruebas de aceptación y pueden ser manuales o automatizadas.
- **Pruebas de Regresión**: Las pruebas de regresión se realizan para **verificar que los cambios realizados en el software no han afectado el funcionamiento de las funcionalidades previamente probadas**. Se ejecutan durante todo el proceso de desarrollo y mantenimiento de software.
- **Pruebas Exploratorias**: Las pruebas exploratorias son pruebas ad-hoc que se realizan **para descubrir errores o problemas que no se habían previsto en las pruebas planificadas**. Estas pruebas se realizan en la etapa de pruebas de sistema o de aceptación.
- **Pruebas de Carga**: Las pruebas de carga se realizan para **verificar el comportamiento del software bajo condiciones de carga extrema**, como picos de tráfico, grandes cantidades de datos, entre otros. Estas pruebas se realizan en la etapa de pruebas de sistema.

Cada tipo de prueba se enfoca en verificar diferentes aspectos del software y se aplica en diferentes etapas del proceso de desarrollo de software. Es importante utilizar una combinación de estos tipos de pruebas para garantizar la calidad del software y reducir la posibilidad de errores.

Características de una buena prueba

Una buena prueba debe cumplir con ciertas características para ser considerada efectiva y confiable. Algunas de las características más importantes que debe tener una buena prueba son las siguientes:

- **Repetibilidad**: Una buena prueba debe ser reproducible, es decir, **debe producir los mismos resultados cada vez que se ejecuta**. Esto asegura que la prueba sea consistente y confiable.
- **Cobertura**: Una buena prueba debe **cubrir una amplia gama de casos de prueba** para garantizar que se prueben todas las funcionalidades relevantes del sistema o componente.
- **Precisión**: La prueba debe ser precisa, es decir, debe **medir exactamente lo que se pretende probar**. Esto implica que la prueba debe ser diseñada con cuidado y atención a los detalles.
- **Independencia**: La prueba debe ser independiente de otras pruebas, lo que significa que **debe ser capaz de ser ejecutada sin depender de otras pruebas**. Esto facilita la detección de errores y la identificación de las causas raíz de los mismos.
- **Facilidad de mantenimiento**: Las pruebas deben ser fáciles de mantener y actualizar. Esto significa que las pruebas deben ser **fáciles de entender y de modificar,** y que deben ser actualizadas para reflejar los cambios en el sistema o en los requisitos.
- **Complejidad adecuada**: Una buena prueba debe ser lo suficientemente **compleja para detectar errores, pero no** tan compleja que sea **difícil de entender o de ejecutar**.
- **Claridad y documentación**: Una buena prueba debe ser **clara y estar bien documentada** para facilitar su comprensión y uso por parte de otros miembros del equipo de desarrollo.

En resumen, las características de una buena prueba incluyen repetibilidad, cobertura, precisión, independencia, facilidad de mantenimiento, complejidad adecuada y claridad y documentación. Una buena prueba debe ser diseñada cuidadosamente y ejecutada de manera rigurosa para garantizar la calidad del software y minimizar los errores en el sistema o componente que se está probando.

Tests Unitario

El testing unitario es una técnica de prueba de software que **se enfoca en comprobar el correcto funcionamiento de las unidades de código** individualmente, como pueden ser clases, métodos o funciones, de forma aislada. Esta técnica de prueba **se lleva a cabo en la etapa de desarrollo** y se ejecuta con frecuencia durante la fase de pruebas de unidad. **El objetivo es detectar errores en el código en una etapa temprana del proceso de desarrollo**, antes de que se realice la integración con otras partes del sistema. Esto permite detectar y corregir errores en el código de manera más rápida y sencilla, **lo que reduce los costos y el tiempo de desarrollo**.

Las pruebas unitarias se realizan utilizando frameworks de pruebas como JUnit y Mockito, que proporcionan un conjunto de herramientas para facilitar la escritura y ejecución de pruebas unitarias. **Estas pruebas suelen ser automatizadas y se ejecutan de forma rápida**, permitiendo una mayor frecuencia de ejecución y una retroalimentación rápida del estado del código.

La implementación de pruebas unitarias **requiere de una metodología y prácticas de desarrollo adecuadas, que incluyen la escritura de código limpio, modular y con alta cohesión**, **el uso de patrones de diseño y buenas prácticas de codificación**. También es necesario realizar una planificación adecuada de las pruebas unitarias, estableciendo una estrategia de pruebas y una estructura de organización de las pruebas.

Herramientas de testing

Junit y Mockito son dos de las herramientas de testing más populares para el lenguaje de programación Java. **Junit es un framework de pruebas unitarias** para Java **que permite** a los desarrolladores **escribir pruebas automatizadas** para sus aplicaciones. **Mockito**, por su parte, **es un framework** de pruebas de integración y unitarias **que se utiliza para simular objetos y comportamientos en las pruebas**.

Junit se utiliza principalmente para pruebas de unidad y es compatible con varios entornos de desarrollo integrado (IDE) como NetBeans y IntelliJ IDEA. Una de las características más útiles de Junit es su capacidad para ejecutar pruebas en paralelo, lo que permite ahorrar tiempo en la ejecución de pruebas y acelerar el ciclo de desarrollo. Además, permite la configuración de pruebas para establecer y verificar el estado esperado de los objetos que se están probando, lo que ayuda a garantizar la precisión y la integridad de las pruebas.

Por otro lado, **Mockito se utiliza principalmente para pruebas de integración y unitarias**. Se utiliza para simular objetos y comportamientos en las pruebas, lo que facilita la escritura de pruebas para objetos complejos. Mockito se integra bien con Junit y otros frameworks de pruebas, y proporciona una sintaxis fácil de usar para simular objetos y comportamientos. Se pueden simular objetos que aún no se han implementado, lo que permite probar el comportamiento de un objeto antes de que se implemente. Además, se puede utilizar para simular objetos que no son fáciles de probar, como bases de datos o servicios web, lo que facilita la escritura de pruebas de integración.

En conclusión, Junit y Mockito son herramientas de testing útiles y populares en el desarrollo de software en Java. Junit se utiliza principalmente para pruebas de unidad y Mockito para pruebas de integración y unitarias. Ambos frameworks proporcionan características útiles para la escritura y ejecución de pruebas automatizadas, lo que ayuda a garantizar la calidad y la integridad del software.

JUNIT

Junit es un framework de pruebas unitarias para el lenguaje de programación Java. Fue creado por Erich Gamma y Kent Beck en 1997, y ha sido adoptado ampliamente por la comunidad de desarrollo de Java debido a su facilidad de uso y capacidad para automatizar pruebas de software.

La idea detrás de Junit es que las pruebas de software deben ser automatizadas y repetibles. Junit proporciona una estructura de marco de trabajo para escribir pruebas unitarias en Java y una serie de métodos y anotaciones para definir pruebas.

El uso de Junit implica la creación de clases de prueba que contienen métodos de prueba que verifican el comportamiento del código bajo prueba. Cada método de prueba debe ser marcado con la anotación "***@Test*"** para indicar que es un método de prueba.

Una vez que se han definido las clases y los métodos de prueba, se puede ejecutar las pruebas utilizando una herramienta de construcción como Maven o Gradle. Junit proporciona una clase de ejecución de pruebas, **JUnitCore**, que se puede utilizar para ejecutar pruebas desde una línea de comando o mediante programación.

Durante la ejecución de pruebas, Junit utiliza el concepto de aserciones para verificar el comportamiento del código bajo prueba. **Las aserciones son**

declaraciones que especifican el estado esperado de los objetos que se están probando. Si una aserción falla durante la ejecución de la prueba, se registra un error y se detiene la ejecución de la prueba.

Además de la anotación "***@Test***", Junit proporciona una serie de otras anotaciones y métodos para personalizar y configurar pruebas, como "***@Before***", que se utiliza para configurar el estado de las pruebas antes de su ejecución, y "***@After***", que se utiliza para limpiar el estado después de la ejecución de las pruebas.

En resumen, Junit es un framework de pruebas unitarias para Java que proporciona una estructura de marco de trabajo para escribir pruebas automatizadas y una serie de métodos y anotaciones para definir pruebas. Junit utiliza aserciones para verificar el comportamiento del código bajo prueba y se puede ejecutar utilizando una herramienta de construcción como Maven o Gradle.

Para explotar al máximo este framework, se recomienda la lectura de la documentación oficial de JUnit 5 User Guide. Durante el cursado de la materia, estudiaremos los conceptos y anotaciones claves para poder hacer un testing efectivo del código que escribiremos durante el desarrollo de la asignatura.

Writing Tests

Aquí se muestra un ejemplo de cómo escribir pruebas en JUnit para una clase simple llamada ***Calculator*** que tiene dos métodos, ***add*** y ***subtract***:

```java
import org.junit.Test;
import static org.junit.Assert.*;

public class CalculatorTest {

    @Test
    public void testAdd() {
        Calculator calc = new Calculator();
        int result = calc.add(2, 3);
        assertEquals(5, result);
    }

    @Test
    public void testSubtract() {
        Calculator calc = new Calculator();
        int result = calc.subtract(5, 3);
        assertEquals(2, result);
    }
}
```

Imagen 1: Elaboración Propia.

En este ejemplo, hemos creado una clase de prueba ***CalculatorTest*** que contiene dos métodos de prueba, ***testAdd*** y ***testSubtract***. Cada método de prueba está marcado con la anotación ***@Test*** para indicar que es una prueba.

Dentro de cada método de prueba, creamos una instancia de la clase ***Calculator*** que estamos probando y llamamos a sus métodos ***add*** y ***subtract*** con valores conocidos. Luego, utilizamos el método ***assertEquals*** para verificar que los resultados devueltos son iguales a los valores esperados.

En este caso, estamos probando que el método ***add*** suma dos números correctamente y que el método ***subtract*** resta dos números correctamente. Si alguna de estas pruebas falla, el método de prueba correspondiente indicará que la prueba ha fallado y proporcionará información sobre por qué la prueba ha fallado.

Es importante tener en cuenta que estos son ejemplos simples y que las pruebas unitarias pueden ser mucho más complejas y abarcar muchos escenarios diferentes. La idea es escribir pruebas que cubran todos los casos posibles y que aseguren que el código que estamos probando funcione correctamente en todas las situaciones.

@Test

Esta anotación se utiliza para identificar métodos que deben ejecutarse como parte de las pruebas unitarias. Cuando se utiliza la anotación ***@Test***, se espera que el método de prueba contenga una o más afirmaciones (***assertions***), **que son expresiones que verifican si el resultado** esperado de una operación es igual al resultado real.

Si una afirmación falla, se informará que el método de prueba ha fallado. Si todas las afirmaciones en el método de prueba son exitosas, se considera que el método de prueba ha pasado.

@DisplayName

La anotación ***@DisplayName*** es una anotación en JUnit 5 que se utiliza para **dar un nombre legible a un método de prueba**. La anotación se coloca encima de la definición del método de prueba y se utiliza para proporcionar un nombre que se utilizará en los informes de prueba. Acepta una cadena como parámetro que especifica el nombre del método de prueba. Esta cadena puede incluir caracteres Unicode y espacios en blanco, lo que la hace muy útil para dar nombres descriptivos y legibles a los métodos de prueba.

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

@DisplayName("A special test case")
class DisplayNameDemo {

    @Test
    @DisplayName("Custom test name containing spaces")
    void testWithDisplayNameContainingSpaces() {
    }

    @Test
    @DisplayName("ノ°□°)ノ")
    void testWithDisplayNameContainingSpecialCharacters() {
    }

    @Test
    @DisplayName("😱")
    void testWithDisplayNameContainingEmoji() {
    }

}
```

Imagen 2: Elaboración Propia.

@BeforeEach

La anotación ***@BeforeEach*** es una anotación en JUnit 5 que se utiliza para **indicar que un método debe ser ejecutado antes de cada método de prueba** en una clase de prueba. Este método se utiliza comúnmente para configurar el estado inicial necesario para los métodos de prueba.

Por ejemplo, si un conjunto de pruebas depende de ciertos objetos que necesitan ser inicializados antes de cada prueba, la anotación ***@BeforeEach*** se puede utilizar para asegurarse de que estos objetos estén inicializados antes de cada prueba.

@AfterEach

La anotación ***@AfterEach*** es una anotación en JUnit 5 que se utiliza para **indicar que un método debe ser ejecutado después de cada método de prueba** en una clase de prueba. Este método se utiliza comúnmente para liberar los recursos que se han utilizado en los métodos de prueba.

Por ejemplo, si un conjunto de pruebas depende de ciertos objetos que necesitan ser liberados después de cada prueba, la anotación @AfterEach se puede utilizar para asegurarse de que estos objetos estén liberados después de cada prueba.

@BeforeAll y @AfterAll

Las anotaciones *@BeforeEach* y *@AfterEach* son anotaciones que se utilizan en JUnit 5 para configurar el estado de cada método de prueba en una clase de prueba. Estas anotaciones se utilizan para asegurarse de que cada método de prueba se ejecuta en un estado predecible y coherente.

Por otro lado, las anotaciones ***@BeforeAll*** y ***@AfterAll*** **son anotaciones que se utilizan en JUnit 5 para configurar el estado inicial y final de una clase de prueba**. Estas anotaciones se utilizan para inicializar y limpiar los recursos que se utilizan en todas las pruebas de la clase.

La anotación ***@BeforeAll*** se utiliza para indicar que un método **debe ser ejecutado una sola vez antes de que se ejecute cualquier método de prueba** en una clase de prueba. Por otro lado, la anotación ***@AfterAll*** se utiliza para indicar que un método **debe ser ejecutado una sola vez después de que se hayan ejecutado todos los métodos** de prueba en una clase de prueba.

@Tag

La anotación ***@Tag*** es una anotación en JUnit 5 que se utiliza para etiquetar pruebas y agruparlas lógicamente. Los tags pueden ser útiles para filtrar y ejecutar un subconjunto de pruebas basándose en la categoría de prueba a la que pertenecen.

La anotación @Tag se utiliza colocando la etiqueta deseada como parámetro de la anotación. Aquí hay un ejemplo:

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

@Tag("fast")
class MyFastTests {

    @Test
    void test1() {
        // Código de prueba
    }

    @Test
    void test2() {
        // Código de prueba
    }
}
```

Imagen 3: Elaboración Propia.

En este ejemplo, la etiqueta "fast" se ha aplicado a la clase de prueba MyFastTests. Esto significa que cualquier prueba en esta clase de prueba que tenga la misma etiqueta será agrupada bajo la categoría "fast". Esto permite la selección y ejecución específica de todas las pruebas que pertenecen a la categoría "fast".

La anotación ***@Tag*** también se puede aplicar a nivel de método de prueba. Aquí hay un ejemplo:

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

class MyTests {

    @Test
    @Tag("fast")
    void test1() {
        // Código de prueba
    }

    @Test
    @Tag("slow")
    void test2() {
        // Código de prueba
    }
}
```

Imagen 4: Elaboración Propia.

En este ejemplo, los métodos de prueba test1() y test2() tienen las etiquetas "fast" y "slow", respectivamente. Esto significa que pueden ser agrupados y ejecutados específicamente como un grupo utilizando la herramienta de línea de comandos de JUnit.

@Disabled

La anotación ***@Disabled*** es una anotación en JUnit 5 que se utiliza para deshabilitar temporalmente una prueba o conjunto de pruebas sin tener que eliminarlas del código fuente. Es útil para pruebas que no están funcionando correctamente, pero que aún no se han eliminado debido a la posibilidad de que se puedan solucionar en el futuro.

Al igual que la anotación @Tag, la anotación ***@Disabled*** se puede aplicar a nivel de clase o de método de prueba.

Standard test class

```java
import static org.junit.jupiter.api.Assertions.fail;
import static org.junit.jupiter.api.Assumptions.assumeTrue;

import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Disabled;
import org.junit.jupiter.api.Test;

class StandardTests {

    @BeforeAll
    static void initAll() {
    }

    @BeforeEach
    void init() {
    }

    @Test
    void succeedingTest() {
    }

    @Test
    void failingTest() {
        fail("a failing test");
    }

    @Test
    @Disabled("for demonstration purposes")
    void skippedTest() {
        // not executed
    }

    @Test
    void abortedTest() {
        assumeTrue("abc".contains("Z"));
        fail("test should have been aborted");
    }

    @AfterEach
    void tearDown() {
    }

    @AfterAll
    static void tearDownAll() {
    }

}
```

Imagen 5: Elaboración Propia.

MOCKITO

Mockito **es una biblioteca de Java utilizada para crear objetos simulados (mocks) en pruebas unitarias**. Fue creada por Szczepan Faber en 2007 como una respuesta a las limitaciones de las bibliotecas de simulación existentes en ese momento.

En pruebas unitarias, a menudo **es necesario simular el comportamiento de objetos o dependencias externas para aislar la funcionalidad que se está probando**. Mockito permite a los desarrolladores crear objetos simulados que se comportan de manera similar a los objetos reales, pero con control adicional sobre su comportamiento.

Mockito utiliza una sintaxis clara y concisa para definir y verificar el comportamiento de los objetos simulados. Los desarrolladores pueden definir lo que se debe devolver cuando se llama a un método simulado, lanzar excepciones y verificar que se llame a un método simulado con los argumentos correctos.

Desde su lanzamiento en 2007, se ha convertido en una de las bibliotecas de simulación de objetos más populares en el ecosistema de Java. Mockito es una biblioteca de código abierto y se puede utilizar en combinación con otras bibliotecas de prueba, como JUnit.

El proceso de utilizar Mockito generalmente consta de tres pasos:

1. Crear un objeto simulado utilizando la clase ***Mockito.mock()***.
2. Definir el comportamiento del objeto simulado utilizando el método ***Mockito.when()***.
3. Verificar que se llame a los métodos del objeto simulado con los argumentos correctos utilizando el método ***Mockito.verify()***.

mock() / @Mock

¿Qué es un Mock?

Un mock es un objeto simulado que se utiliza en lugar de un objeto real. Se crean para simular el comportamiento de una dependencia externa o un objeto complejo que no se puede o no se quiere utilizar en una prueba unitaria.

Los objetos simulados **se utilizan para aislar la funcionalidad que se está probando y para controlar el comportamiento de la dependencia externa**. Por ejemplo, en una prueba unitaria de una clase que realiza operaciones de base de datos, se podría crear un objeto simulado que represente la base de datos en lugar

de utilizar una base de datos real. Se puede definir lo que se debe devolver cuando se llama a un método simulado, lanzar excepciones y verificar que se llame a un método simulado con los argumentos correctos.

En Mockito, el método ***mock()*** o la anotación ***@Mock*** se utilizan para crear un objeto simulado (mock object) de una clase o interfaz en particular. Este objeto simulado se puede utilizar para simular el comportamiento de un objeto real durante la ejecución de pruebas unitarias.

La forma más común de crear un objeto simulado es usando la anotación ***@Mock***. Esta anotación se coloca encima de la declaración de una variable de instancia y le indica a Mockito que se debe crear un objeto simulado de la clase o interfaz correspondiente.

```
public class MyClassTest {
    @Mock
    private MyDependency myDependency;
    ...
}
```

Imagen 6: Elaboración Propia.

También se puede crear un objeto simulado utilizando el método mock(), que es parte de la clase Mockito. Este método toma como argumento una clase o interfaz y devuelve un objeto simulado de esa clase o interfaz.

```
MyDependency myDependency = Mockito.mock(MyDependency.class);
```

Imagen 7: Elaboración Propia.

Una vez que se ha creado un objeto simulado, se puede configurar su comportamiento utilizando métodos como ***when()*** y ***thenReturn()***

spy() / @Spy

¿Qué es un Spy?

En Mockito, un "spy" **es un objeto simulado que se crea a partir de un objeto real**. A diferencia de un objeto simulado creado con mock(), un objeto **spy mantiene el comportamiento original del objeto real, pero también se puede configurar para hacer cosas adicionales, como registrar llamadas a métodos y modificar el valor de retorno.**

El método ***spy()*** permite crear un objeto "spy", el cual es un objeto simulado que se crea a partir de un objeto real. Para utilizar spy(), se debe pasar un objeto real como parámetro. Por ejemplo:

```
List<String> myList = new ArrayList<>();
List<String> mySpy = Mockito.spy(myList);
```

Imagen 8: Elaboración Propia.

En este ejemplo, se crea un objeto myList de la clase ArrayList, y luego se crea un objeto mySpy que es un spy de myList. Ahora se puede utilizar el objeto mySpy en lugar de myList durante la ejecución de la prueba unitaria.

Por otro lado, la anotación ***@Spy*** se utiliza para crear un objeto spy utilizando la inyección de dependencias. Por ejemplo:

```
public class MyClassTest {
  @Spy
  private MyClass mySpy;

  // ...
}
```

Imagen 9: Elaboración Propia.

En este ejemplo, se crea un objeto mySpy que es un spy de MyClass, y se utiliza la anotación ***@Spy*** para indicar que se debe crear el objeto utilizando la inyección de dependencias. Una vez que se ha creado un objeto spy, se puede configurar su comportamiento utilizando los mismos métodos que se utilizan para configurar el comportamiento de un objeto simulado creado con mock().

@InjectMocks

La anotación ***@InjectMocks*** se utiliza para **inyectar automáticamente los mocks y spies creados con las anotaciones @Mock y @Spy respectivamente en una instancia de la clase que estamos probando**.

Al utilizar esta anotación, Mockito buscará en la instancia de la clase que estamos probando, cualquier campo anotado con @Mock o @Spy y los inyectará en esa instancia para que puedan ser utilizados durante las pruebas.

Es importante tener en cuenta que @InjectMocks solo puede ser utilizada para inyectar mocks o spies en una instancia de una clase, por lo que no funciona con objetos creados mediante new. Además, Mockito utilizará su algoritmo interno para resolver las dependencias, lo que significa que no siempre se puede garantizar que se utilicen las mismas instancias de los mocks y spies en todas las pruebas. Aquí hay un ejemplo de cómo se puede utilizar la anotación @InjectMocks:

```
public class MyClassTest {

    @Mock
    private MyDependency myDependency;

    @InjectMocks
    private MyClass myClass;

    @Test
    public void testMyMethod() {
        // Configurar el comportamiento del mock
        when(myDependency.doSomething()).thenReturn("Resultado esperado");

        // Ejecutar el método que estamos probando
        String result = myClass.myMethod();

        // Verificar el resultado
        assertEquals("Resultado esperado", result);
    }
}
```

Imagen 10: Elaboración Propia.

En este ejemplo, se ha creado un mock llamado myDependency y se ha anotado la instancia de MyClass que se está probando con @InjectMocks. Al ejecutar la prueba, Mockito inyectará automáticamente el mock myDependency en la instancia de MyClass que se está probando, lo que permitirá probar el método myMethod() de MyClass con el comportamiento simulado del mock myDependency.

when()

En Mockito, el método when() se utiliza para **configurar el comportamiento de un objeto simulado o spy** cuando se llama a un método en particular. Por ejemplo:

```
List<String> myList = mock(List.class);
when(myList.size()).thenReturn(3);
```

Imagen 11: Elaboración Propia.

En este ejemplo, se crea un objeto simulado de la clase List utilizando el método mock(). Luego, **se utiliza el método when() para configurar el comportamiento del método size(), indicando que debe devolver 3 cuando se llame.**

Cuando se llama al método size() en el objeto simulado, Mockito intercepta la llamada y devuelve el valor configurado en la instrucción thenReturn(), en lugar de ejecutar el método real. Esto permite controlar el comportamiento del objeto simulado durante la ejecución de la prueba unitaria. El método when() también se puede utilizar para configurar el comportamiento de un método para que lance una excepción en lugar de devolver un valor. Por ejemplo:

```
List<String> myList = mock(List.class);
when(myList.get(0)).thenThrow(new IndexOutOfBoundsException());
```

Imagen 12: Elaboración Propia.

En este ejemplo, se configura el comportamiento del método get() para que lance una excepción IndexOutOfBoundsException cuando se llame con el índice 0.

Es importante tener en cuenta que para utilizar el método when(), es necesario haber creado previamente un objeto simulado o spy utilizando los métodos mock() o spy().

thenReturn()

En Mockito, el método ***thenReturn()*** se utiliza para configurar **el valor de retorno de un método en un objeto simulado o spy**. Tal como vimos en el ejemplo anterior. La sintaxis del método thenReturn() es la siguiente:

```
when(mock.method()).thenReturn(value);
```

Imagen 13: Elaboración Propia.

Donde mock es el objeto simulado o spy, method es el método al que se le está configurando el valor de retorno y value es el valor que debe devolver el método cuando se llama. Es importante tener en cuenta que el valor configurado con thenReturn() solo se aplica cuando se llama al método en cuestión en el objeto simulado o spy. Si se llama a cualquier otro método en el objeto, se utilizará el comportamiento predeterminado de Mockito, que suele ser devolver null, false o 0, según el tipo de dato de retorno del método.

thenThrow()

En Mockito, el método ***thenThrow()*** se utiliza para configurar **una excepción que se lanzará cuando se llame a un método** en un objeto simulado o spy, tal como se mostró en el primer ejemplo. La sintaxis del método thenThrow() es la siguiente:

```
when(mock.method()).thenThrow(exception);
```

Imagen 14: Elaboración Propia.

Al igual que antes, mock es el objeto simulado o spy, method es el método al que se le está configurando la excepción y exception es la excepción que se lanzará cuando se llama al método.

doNothing()

En Mockito, el método ***doNothing()*** se utiliza para configurar un objeto simulado o spy **para que no haga nada cuando se llama a un método en particular**. Este método se utiliza principalmente en pruebas en las que se desea ignorar ciertas partes del código y asegurarse de que no se están ejecutando.

Por ejemplo, supongamos que tenemos una clase EmailService que tiene un método sendEmail() que envía un correo electrónico a un destinatario determinado. Si deseamos probar una clase que depende de EmailService, pero no queremos enviar correos electrónicos reales, podemos utilizar un objeto simulado o spy de EmailService y configurar el método sendEmail() para que no haga nada utilizando el método doNothing() de Mockito. La sintaxis del método doNothing() es la siguiente:

```
doNothing().when(mock).method();
```

Imagen 15: Elaboración Propia.

Y al igual que con los otros métodos, mock es el objeto simulado o spy y method es el método al que se le está configurando para no hacer nada.

verify()

En Mockito, el método ***verify()*** se utiliza para comprobar **si un método de un objeto simulado o spy ha sido llamado durante la ejecución de una prueba**. Este método es útil para asegurarse de que ciertas partes del código se están ejecutando correctamente y se están llamando a los métodos adecuados. La sintaxis básica del método verify() es la siguiente:

```
verify(mock, times(n)).method(args);
```

Imagen 16: Elaboración Propia.

Donde mock es el objeto simulado o spy, method es el método que se quiere verificar, times(n) es el número de veces que se espera que se llame al método (por ejemplo, times(1) para verificar que se llamó una sola vez), y args son los argumentos que se pasaron al método (pueden ser nulos).

Por ejemplo, supongamos que tenemos una clase Calculator que tiene un método add() que suma dos números. Si queremos comprobar que se está llamando correctamente al método add() en una clase de prueba, podemos utilizar un objeto simulado o spy de Calculator y el método verify() para comprobarlo.

```
Calculator calculator = mock(Calculator.class);
int result = calculator.add(2, 3);
verify(calculator, times(1)).add(2, 3);
```

Imagen 17: Elaboración Propia.

En este ejemplo, utilizamos el objeto simulado calculator para llamar al método add() y luego usamos el método verify() para comprobar que se llamó al método add() una sola vez con los argumentos 2 y 3.

También se pueden utilizar otras opciones con el método verify(), como ***never()*** para asegurarse de que el método nunca fue llamado, ***atLeast()*** y ***atMost()*** para especificar un rango de llamadas esperadas, entre otros. En general, el método verify() es una herramienta poderosa para asegurarse de que ciertos métodos se están llamando correctamente y garantizar el correcto funcionamiento de la aplicación.

Pruebas en métodos privados

En JUnit, los métodos privados no se pueden probar directamente desde un caso de prueba, ya que no son accesibles fuera de la clase en la que se definen. Sin embargo, hay algunas formas de probar métodos privados en Java. Una forma es utilizar reflexión para acceder a los métodos privados desde el caso de prueba.

Java proporciona la API de reflexión para acceder a los miembros de una clase, incluidos los métodos privados. Para usar la reflexión para acceder a un método privado, puede hacer lo siguiente:

1. Obtener la clase que contiene el método privado.
2. Obtener una referencia al método privado.
3. Establecer el método accesible.
4. Invocar el método privado.

Reflactions

Reflection es una técnica que **permite a un programa inspeccionar y manipular objetos en tiempo de ejecución**, incluso si no se conoce su tipo en tiempo de compilación. Con Reflection, se puede examinar la estructura de una clase (por ejemplo, sus campos, métodos y constructores) y manipular sus objetos (por ejemplo, crear una instancia de una clase, invocar un método en un objeto, etc.).

Reflection se usa comúnmente en bibliotecas y frameworks de Java para lograr la flexibilidad y extensibilidad. Por ejemplo, un framework de inyección de dependencias puede usar Reflection para crear objetos de clase y establecer sus dependencias en tiempo de ejecución o un framework de Test puede usarlo para facilitar el testing de métodos privados sin que esto implique modificar la definición de la clase en tiempo de desarrollo, es decir, podemos acceder a un método privado, cambiar su accesibilidad de private a public e invocarlo solo para esa ejecución sin que esto afecte al resto de las ejecuciones del método.

Si bien la reflexión puede ser útil en algunos casos, es importante tener en cuenta que el acceso a campos y métodos privados a través de la reflexión puede ser considerado una mala práctica de programación ya que rompe el encapsulamiento y puede generar problemas de mantenimiento y legibilidad en el código. Por lo tanto, es importante utilizar la reflexión con precaución y sólo cuando no haya otra alternativa para realizar una prueba unitaria.

ReflectionSupport (JUnit)

La clase ***ReflectionSupport*** del paquete *org.junit.platform.commons.support* de JUnit es una utilidad que proporciona métodos para trabajar con reflexión en el contexto de pruebas unitarias. Esta clase es parte de la plataforma de JUnit y se puede utilizar en cualquier versión de JUnit que utilice la plataforma de JUnit 5.

Aquí hay algunos ejemplos de cómo se puede utilizar esta clase:

Invocar un método privado en una clase

Supongamos que tiene una clase ***MiClase*** que tiene un método privado llamado ***metodoPrivado()*** que desea invocar en una prueba. Puede hacerlo de la siguiente manera:

```
import org.junit.platform.commons.support.ReflectionSupport;

MiClase instancia = new MiClase();

ReflectionSupport.invokeMethod(
    ReflectionSupport.findMethod(MiClase.class, "metodoPrivado"),
    instancia);
```

Imagen 18: Elaboración Propia.

En este ejemplo, utilizamos el método ***invokeMethod*** de ReflectionSupport para invocar el método privado metodoPrivado en la instancia de MiClase. Primero, utilizamos el método ***findMethod*** de ReflectionSupport para buscar el método privado metodoPrivado en la clase MiClase. Luego, pasamos el método y la instancia a invokeMethod para invocar el método.

Obtener el valor de un campo privado en una clase

Supongamos que tiene una clase ***MiClase*** que tiene un campo privado llamado ***campoPrivado*** que desea obtener en una prueba. Puede hacerlo de la siguiente manera:

```
import org.junit.platform.commons.support.ReflectionSupport;

MiClase instancia = new MiClase();

Object valor = ReflectionSupport.getFieldValue(
    ReflectionSupport.findField(MiClase.class, "campoPrivado"),
    instancia);
```

Imagen 19: Elaboración Propia.

En este ejemplo, utilizamos el método ***getFieldValue*** de ReflectionSupport para obtener el valor del campo privado campoPrivado en la instancia de MiClase. Primero, utilizamos el método ***findField*** de ReflectionSupport para buscar el campo privado campoPrivado en la clase MiClase. Luego, pasamos el campo y la instancia a ***getFieldValue*** para obtener el valor del campo.

findMethod()

El método findMethod se utiliza para buscar un método en una clase utilizando su nombre y parámetros. El método acepta dos argumentos:

- **Class<?> clazz**: La clase en la que se va a buscar el método.
- **String methodName**: El nombre del método que se desea buscar.
- **Opcionalmente**, también se pueden especificar **los tipos de los parámetros del método** utilizando un arreglo de objetos Class<?> como tercer argumento.

El método devuelve un objeto ***Optional<Method>, lo que significa que el valor devuelto puede estar presente o no,*** es decir, que representa la opción de haber encontrado el método buscado, o un opcional vacío si no se encuentra ningún método que coincida con el nombre y los parámetros especificados.

invokeMethod()

El método invokeMethod se utiliza para invocar un método en una instancia de una clase. El método acepta tres argumentos:

- **Object instance**: La instancia de la clase en la que se va a invocar el método.
- **Method method**: El objeto Method que representa el método que se va a invocar.
- **Object... args**: Los argumentos que se van a pasar al método (opcional).

El método devuelve un objeto **Object** que representa el resultado de la invocación del método. Si el método no devuelve ningún valor (es decir, su tipo de retorno es void), el método invokeMethod devuelve null.

findField()

El método findField se utiliza para buscar un campo (o atributo) en una clase. El método acepta dos argumentos:

- **Class<?> clazz**: La clase en la que se va a buscar el campo.
- **String fieldName**: El nombre del campo que se va a buscar.

El método devuelve un objeto **Optional<Field>** que representa el campo buscado. Si el campo no se encuentra, el objeto Optional está vacío.

getFieldValue()

El método getFieldValue se utiliza para obtener el valor de un campo (o atributo) de una instancia de una clase. El método acepta dos argumentos:

- **Object instance**: La instancia de la clase en la que se encuentra el campo.
- **Field field**: El objeto Field que representa el campo del que se desea obtener el valor.

El método devuelve un objeto de tipo **Object** que representa el valor del campo en la instancia de la clase especificada.

setFieldValue()

El método setFieldValue se utiliza para establecer el valor de un campo (o atributo) de una instancia de una clase. El método acepta tres argumentos:

- **Object instance**: La instancia de la clase en la que se encuentra el campo.
- **Field field**: El objeto Field que representa el campo del que se desea establecer el valor.
- **Object value**: El valor que se desea establecer en el campo.

El método no devuelve ningún valor.

BIBLIOGRAFÍA

- Página oficial de JUnit (https://junit.org/).
- Página oficial de Mockito (https://site.mockito.org/).