





Mockito



mockito



Mockito

Mockito es un mocking framework, una biblioteca basada en Java que se utiliza para realizar pruebas unitarias efectivas de las aplicaciones Java. Mockito se utiliza para simular interfaces, de modo que se puede agregar una funcionalidad ficticia a una interfaz simulada que se puede usar en pruebas unitarias.

Mockito

Mockito facilita la creación de objetos simulados. Utiliza Java Reflection para crear objetos simulados para una interfaz determinada.

Consideremos un caso de Stock Service que devuelve los detalles del precio de un producto. Durante el desarrollo, el servicio de stock real no se puede utilizar para obtener datos en tiempo real. Así que necesitamos una implementación ficticia del servicio de stock. Esto se puede hacer con Mockito.

Mockito

El mocking es una forma de probar la funcionalidad de una clase aislada. El mock no requiere una conexión a la base de datos, un archivo de propiedades leído o un servidor de archivos leído para probar una funcionalidad. Los objetos simulados hacen mocking del servicio real. Un objeto simulado devuelve datos ficticios correspondientes a alguna entrada ficticia que se le pasa.

Mockito

Ventajas de usar Mockito:

- Sin escritura a mano: no es necesario escribir objetos simulados por tu cuenta.
- Refactoring Safe: el cambio de nombre de los nombres de los métodos de la interfaz o la reordenación de los parámetros no romperá el código de prueba, ya que los Mocks se crean en tiempo de ejecución.
- Compatibilidad con el valor de retorno: admite valores de retorno.

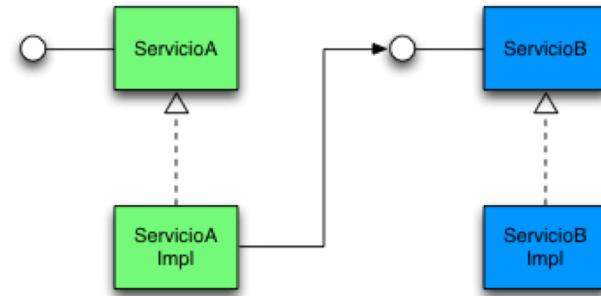
Mockito

Ventajas de usar Mockito:

- Soporte de excepciones: admite excepciones.
- Compatibilidad con verificación de pedidos: admite la verificación en orden de llamadas de método.
- Soporte de anotación: admite la creación de simulacros mediante la anotación.

Mockito

Java Mockito es uno de los frameworks de Mock más utilizados en la plataforma Java. Esto es debido a la gran facilidad de uso que tiene. Vamos a explicar el concepto de Mock y como funciona Java Mockito , para ello construiremos unos test sencillos partiendo de dos clases de Servicio que se encuentran relacionadas.



Mockito

En este caso una clase se encarga de sumar dos números y la otra se encarga de multiplicarlos.

```
public interface ServicioA {  
    public abstract int sumar(int a, int b);  
}
```

Mockito

```
public class ServicioAImpl implements ServicioA {  
    public int sumar(int a ,int b) {  
        return a+b;  
    }  
}
```

Mockito

```
public interface ServicioB {  
    public ServicioA getServicioA();  
    public void setServicioA(ServicioA servicioA);  
    public int multiplicarSumar(int a, int b, int multiplicador);  
    public int multiplicar(int a, int b);
```

Mockito

```
public class ServicioBImpl implements ServicioB {  
    private ServicioA servicioA;  
    public ServicioA getServicioA() {  
        return servicioA;  
    }  
  
    public void setServicioA(ServicioA servicioA) {  
        this.servicioA = servicioA;  
    }  
  
    public int multiplicarSumar(int a ,int b,int multiplicador) {  
        return servicioA.sumar(a, b)*multiplicador;  
    }  
}
```

Mockito

```
public int multiplicar(int a ,int b) {  
  
    return a*b;  
}  
}
```

Mockito

Como podemos observar ambas clases disponen de un método que se puede testear de forma aislada ServicioA (sumar) y ServicioB (multiplicar). Vamos a instalar JUnit y Mockito como dependencias de Maven para poder trabajar.

Mockito

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>net.impacto.mockito</groupId>
<artifactId>mocks</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>mocks</name>
<dependencies>
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>4.12</version>
</dependency>
```

Mockito

```
<dependency>
<groupId>org.mockito</groupId>
<artifactId>mockito-core</artifactId>
<version>1.10.19</version>
</dependency>
</dependencies>
</project>
```

Mockito

Construimos los test :

```
import org.junit.Assert;
import org.junit.Test;
import net.impacto.mockito.ServicioA;
import net.impacto.mockito.ServicioAImpl;
public class TestServicioA {
    @Test
    public void testSuma() {
        int a=2;
        int b=2;
        ServicioA servicio= new ServicioAImpl();
        Assert.assertEquals(4, servicio.sumar(a, b));
    }
}
```

Mockito

Construimos los test :

```
import org.junit.Assert;
import org.junit.Test;

import net.impacto.mockito.ServicioA;
import net.impacto.mockito.ServicioAImpl;
import net.impacto.mockito.ServicioB;
import net.impacto.mockito.ServicioBImpl;

public class TestServicioB {

    @Test
    public void testMultiplicacion() {

        ServicioB servicioB= new ServicioBImpl();
        Assert.assertEquals(servicioB.multiplicar(2, 3),6);

    }
}
```

Mockito

Ejecutamos ambos Test con JUnit.

Ahora bien ¿qué pasará si tenemos que testear el método `sumarMultiplicar` que tiene el `ServicioB`? Este método primero suma dos números delegando en el `ServicioA` y luego los multiplica.

```
@Test  
public void testmultiplicarSumar() {  
  
    ServicioA servicioA=new ServicioAImpl();  
  
    ServicioB servicioB= new ServicioBImpl();  
  
    servicioB.setServicioA(servicioA);  
    Assert.assertEquals(servicioB.multiplicarSumar(2, 3, 2),10);  
  
}
```

Mockito

Todo funciona correctamente. Los problemas aparecen cuando modificamos el método sumar del ServicioA y le asignamos el siguiente código:

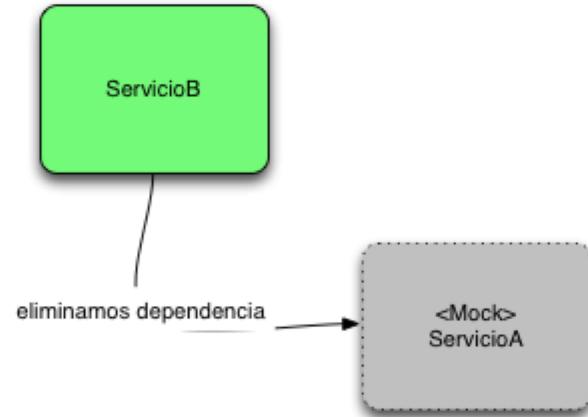
```
package net.impacto.mockito;

public class ServicioAImpl implements ServicioA {
    public int sumar(int a ,int b) {

        return a+b+1;
    }
}
```

Mockito

Fallan los test, esto es un problema importante ya no sabemos que esta pasando exactamente. Para evitar este tipo de problemas debemos aislar las pruebas unitarias y una de las opciones es usar Java Mockito y crear un Mock Object. Los objetos Mock nos permiten simular ser un objeto real y eliminan dependencias, permitiendo que los test se ejecuten de forma aislada.



Mockito

```
@Test  
public void testmultiplicarSumar() {  
  
    ServicioA servicioA=mock(ServicioA.class);  
    when(servicioA.sumar(2,3)).thenReturn(5);  
  
    ServicioB servicioB= new ServicioBImpl();  
    servicioB.setServicioA(servicioA);  
    Assert.assertEquals(servicioB.multiplicarSumar(2, 3, 2),10);  
  
}
```

Hemos creado un mock que simula ser el ServicioA para la operación de suma(2,3) y devuelve como resultado 5.

Mockito

Un test unitario debe probar un componente aislado de nuestra aplicación, los errores o efectos secundarios de otros componentes deben ser eliminados. La pregunta normal sería ¿Cómo aislar mi componente de sus dependencias para así probarlo? Veamos lo que tenemos a nuestra disposición:

Mockito

- Dummy objects : Este tipo de objetos son asignados al componente pero nunca se llaman y por lo general están vacíos.
- Fake objects: Este tipo de objetos tienen implementaciones funcionales pero simplificadas, normalmente utilizan datos que no vienen de la base de datos principal, sino que se tienen en caché u otra fuente más simple.
- Stub classes : Es una implementación parcial de una interfaz o clase con el propósito de utilizarla durante el test, normalmente solo cuentan con los métodos implementados que serán utilizados durante el test.
- Mock objects: Es una implementación Dummy de una interfaz o clase en la cual se define la salida que tendrá la llamada de un método.

Mockito

Veamos otro ejemplo práctico:

DataService.java

```
public interface DataService {  
    int[] getListOfNumbers();  
}
```

Mockito

Veamos otro ejemplo práctico:

CalculatorService.java

```
public interface CalculatorService {  
    double calculateAverage();  
}
```

Mockito

Veamos otro ejemplo práctico:

CalculatorServiceImpl.java

```
public class CalculatorServiceImpl implements CalculatorService {  
    private DataService dataService;  
    @Override  
    public double calculateAverage() {  
        int[] numbers = dataService.getListOfNumbers();  
        double avg = 0;  
        for (int i : numbers) {  
            avg += i;  
        }  
        return (numbers.length > 0) ? avg / numbers.length : 0;  
    }  
  
    public void setDataService(DataService dataService) {  
        this.dataService = dataService;  
    }  
}
```

Mockito

En este caso no escribiremos implementación del DataService ya que haremos un mock del mismo para nuestras pruebas.

Probando nuestro componente

Una vez escritas las implementaciones de nuestros servicios el siguiente punto es probarlas, para esto crearemos un test unitario junto con mocks que permitan simular diferentes respuestas basadas en los tests, veamos el test de ejemplo:

Mockito

```
import static org.mockito.Mockito.when;  
  
import org.junit.Test;  
import org.junit.runner.RunWith;  
import org.mockito.InjectMocks;  
import org.mockito.Mock;  
import org.mockito.junit.MockitoJUnitRunner;  
  
import net.impacto.mockito2.DataService;  
import net.impacto.mockito2.impl.CalculatorServiceImpl;  
  
@RunWith(MockitoJUnitRunner.class)  
public class CalculatorServiceTest {  
    @InjectMocks  
    private CalculatorServiceImpl calculatorService;
```

Mockito

```
@Mock  
private DataService dataService;  
  
@Test  
public void testCalculateAvg_simpleInput() {  
    when(dataService.getListOfNumbers()).thenReturn(new int[] { 1, 2, 3, 4, 5 });  
    assertEquals(3.0, calculatorService.calculateAverage(), .01);  
}
```

Mockito

```
@Test  
public void testCalculateAvg_emptyInput() {  
    when(dataService.getListOfNumbers()).thenReturn(new int[] {});  
    assertEquals(0.0, calculatorService.calculateAverage(), .01);  
}  
  
@Test  
public void testCalculateAvg_singleInput() {  
    when(dataService.getListOfNumbers()).thenReturn(new int[] { 1 });  
    assertEquals(1.0, calculatorService.calculateAverage(), .01);  
}  
}
```

Mockito

Del código anterior podemos analizar lo siguiente:

@RunWith(MockitoJUnitRunner.class) : Se utiliza para definir que se utilizará el Runner de Mockito para ejecutar nuestras pruebas.

@Mock : Se utiliza para informar a Mockito que un objeto mock será injectado en la referencia `dataService`, como se puede ver no es necesario escribir la clase implementación, Mockito lo hace por nosotros.

@InjectMocks : Se utiliza para informar a Mockito que los mocks serán injectados en el servicio definido, es necesario que se cuente con métodos setters en los que se coloca.

`when(dataService.getListOfNumbers()).thenReturn(new int[] { 1, 2, 3, 4, 5 })` : Define que cuando se ejecute el método `getListOfNumbers` se devolverá el resultado `new int[] { 1, 2, 3, 4, 5 }`.

Con lo anterior seremos capaces de probar nuestros componentes de forma aislada asegurando que las condiciones de nuestros tests siempre serán las mismas.

