Pruebas de integración

Dobles de prueba con Mockito

Mockito es una de las bibliotecas más utilizadas para la creación de pruebas unitarias en Java. Permite simular objetos y comportamientos para probar métodos sin depender de implementaciones reales. Vamos a continuar el proyecto anterior y añadirle soporte para Mockito.

> **Actividad**
> Realiza una memoria de todo el proceso de configuración de Mockito

En el archivo `pom.xml, añade esto en `:

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>LATEST</version>
</dependency>
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-junit-jupiter</artifactId>
    <version>LATEST</version>
    <scope>test</scope>
</dependency>
```

A continuación, debemos **refrescar** el proyecto para descargar las dependencias. Haz click derecho en el archivo `pom.xml` y ve a Maven->Sync Project.

Supongamos que tenemos una clase `CalculadoraService` que depende de un `Repositorio` para obtener datos. Añadamos la clase a nuestro proyecto en un archivo `CalculadoraService` dentro de src/main/java:

```
public class CalculadoraService {
    private final Repositorio repositorio;

    public CalculadoraService(Repositorio repositorio) {
        this.repositorio = repositorio;
    }

    public int sumarValores() {
        return repositorio.obtenerValorA() + repositorio.obtenerValorB();
    }
    //Lo usaremos más tarde en las opciones avanzadas de mockito
    public int sumarValores(int a, int b){
        return a + b;
    }
}
```

El repositorio nos va a salir en rojo, así que creamos una interfaz para representar los métodos que necesitaremos de él en un archivo `Repositorio.java` :

```
public interface Repositorio {
    int obtenerValorA();
    int obtenerValorB();
}
```

Al haberlo creado en el mismo paquete, no es necesario añadir ninguna línea de importación. Si lo hubiéramos creado en otro paquete distinto, deberíamos importar las clases de forma adecuada, empleando las herramientas que nos proporciona IntelliJ, de forma similar a como hicimos al importar la etiqueta `@Test` en el apartado anterior.

Lo siguiente es crear un test usando Mockito. Para ello, generamos los test de `CalculadoraService` igual que en el apartado anterior.

1. En `src/test/java` , crea una clase de prueba:

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

    @Mock
    private Repositorio repositorio;

    //Esto inserta el Mock anterior como parámetro en el constructor de CalculadoraServ
    @InjectMocks
    private CalculadoraService calculadoraService;

    @Test
    void testSumarValores() {
        when(repositorio.obtenerValorA()).thenReturn(5);
        when(repositorio.obtenerValorB()).thenReturn(3);

        int resultado = calculadoraService.sumarValores();

        assertEquals(8, resultado);
    }
}
```

Volverán a aparecer elementos sin importar. Impórtalos todos haciendo uso de la herramienta de IntelliJ para tal efecto.

- `@Mock` simula un objeto `Repositorio`.
- `@InjectMocks` inyecta el mock en `CalculadoraService`.
- `when(...).thenReturn(...)` define valores falsos que devolverá el mock.
- `assertEquals(8, resultado)` verifica el resultado esperado.

Ejecutar la prueba

- Haz clic derecho en la prueba y selecciona **Run 'CalculadoraServiceTest'**.
- Verifica que la prueba pase correctamente.

Cuando ejecutes el código, funcionará bien pero te saldrá un Warning:

```
Mockito is currently self-attaching to enable the inline-mock-maker. This will no longe
WARNING: A Java agent has been loaded dynamically (/Users/arturoalbero/.m2/repository/n
WARNING: If a serviceability tool is in use, please run with -XX:+EnableDynamicAgentLoa
WARNING: If a serviceability tool is not in use, please run with -Djdk.instrument.trace
WARNING: Dynamic loading of agents will be disallowed by default in a future release
OpenJDK 64-Bit Server VM warning: Sharing is only supported for boot loader classes bec
```

Esto sucede porque Mockito carga un agente de forma dinámica, comportamiento que se pretende restrigir en futuras versiones de Java. En este caso, se está usando la versión 21. Para solucionar los WARNING (o futuros errores), debes añadir después de `</dependencies>` el siguiente apartado en tu archivo `pom.xml` :

```xml
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>3.0.0-M7</version>
            <configuration>
                <argLine>-javaagent:"${settings.localRepository}/net/bytebuddy/byte-bud
            </configuration>
        </plugin>
    </plugins>
</build>
```

Lo que estamos haciendo es añadir Mockito como un agente a nuestro proyecto Maven, tal y como nos decía el *warning*. Refresca el proyecto y ejecuta de nuevo. Te volverá a salir un warning (la última línea), pero de menor importancia:

```
OpenJDK 64-Bit Server VM warning: Sharing is only supported for boot loader classes bec
```

Recuerda que cada vez que hagas un cambio en el archivo `pom.xml` , debes sincronizar el proyecto para que haga efecto.

A continuación, vamos a retocar el código anterior añadiendo todas las funcionalidades de Mockito:

- **Verificación de llamadas** (`verify`)
- **Manejo de excepciones** (`doThrow`)
- **Retornos múltiples** (`thenReturn`)

- **Respuesta personalizada** (`thenAnswer`)
- **Spies para llamar a métodos reales de la clase si es necesario** (`@Spy`)

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

    @Mock
    private Repositorio repositorio;

    //Inyectamos el mock en calculadoraService
    @InjectMocks
    private CalculadoraService calculadoraService;

    //Inyectamos el mock en el espía calculadoraSpy
    @InjectMocks
    @Spy
    private CalculadoraService calculadoraSpy;

    @Test
    void testSumarValores() {
        when(repositorio.obtenerValorA()).thenReturn(5);
        when(repositorio.obtenerValorB()).thenReturn(3);

        int resultado = calculadoraService.sumarValores();

        assertEquals(8, resultado);
        verify(repositorio).obtenerValorA();
        verify(repositorio).obtenerValorB();
    }

    //En este test vamos a probar la obtención de diferentes valores consecutivos.
    @Test
    void testSumarValoresConMultiplesRetornos() {
        when(repositorio.obtenerValorA()).thenReturn(5, 10);
        when(repositorio.obtenerValorB()).thenReturn(3);

        assertEquals(8, calculadoraService.sumarValores());
        assertEquals(13, calculadoraService.sumarValores());
    }

    @Test
    void testSumarValoresConExcepcion() {
        when(repositorio.obtenerValorA()).thenThrow(new RuntimeException("Error al obte

        Exception exception = assertThrows(RuntimeException.class, () -> {
            calculadoraService.sumarValores();
```

```
        });

        assertEquals("Error al obtener valor A", exception.getMessage());
    }

    //En este test verificamos que las operaciones del repositorio se realizan en el or
    @Test
    void testVerificacionDeOrden() {
        when(repositorio.obtenerValorA()).thenReturn(5);
        when(repositorio.obtenerValorB()).thenReturn(3);

        calculadoraService.sumarValores();

        InOrder inOrder = inOrder(repositorio);
        inOrder.verify(repositorio).obtenerValorA();
        inOrder.verify(repositorio).obtenerValorB();
    }

    //En este test, hacemos que el espía mantenga funcionalidades del objeto, y mockeam
    // indicados.
    @Test
    void testUsoDeSpy() {
        doReturn(15).when(calculadoraSpy).sumarValores();

        int resultado = calculadoraSpy.sumarValores();
        assertEquals(15, resultado);
        assertEquals(32, calculadoraService.sumarValores(17, 15));
    }
}
```

Como siempre, debemos importar las dependencias que falten para que funcione.

Principales anotaciones de Mockito:

Anotación	Descripción
`@Mock`	Crea un objeto falso (mock).
`@InjectMocks`	Inyecta automáticamente los mocks en la clase bajo prueba.
`@ExtendWith(MockitoExtension.class)`	Habilita Mockito en JUnit 5.

Anotación	Descripción
`@Spy`	Crea un espía que permite llamar métodos reales y simular comportamientos.
`@Captor`	Crea un `ArgumentCaptor` para capturar valores pasados a métodos mockeados.

Actividad: Pruebas en una aplicación bancaria

1. **Configura un nuevo proyecto en IntelliJ IDEA** con soporte para Java y Maven.
2. **Agrega las dependencias necesarias** en el archivo `pom.xml` para JUnit y Mockito.
3. **Crea una clase** `Banco` con los siguientes métodos:

- `depositar(String cuenta, double monto)` : Aumenta el saldo de una cuenta.
- `retirar(String cuenta, double monto)` : Disminuye el saldo de una cuenta si hay fondos suficientes, de lo contrario, lanza una excepción.
- `consultarSaldo(String cuenta)` : Devuelve el saldo actual de la cuenta.

4. **Define una interfaz** `RepositorioBanco` que represente el acceso a los datos del banco con los métodos:

- `obtenerSaldo(String cuenta)` : Devuelve el saldo de una cuenta.
- `actualizarSaldo(String cuenta, double nuevoSaldo)` : Modifica el saldo de una cuenta.

5. **Implementa** `BancoService` que use `RepositorioBanco` para interactuar con los datos.
6. **Escribe pruebas unitarias con JUnit** para validar el comportamiento de `Banco` .
7. **Usa Mockito para simular** `RepositorioBanco` en las pruebas de `BancoService` .
8. **Amplía las pruebas** aplicando:
    - **Verificación de llamadas** (`verify`)
    - **Manejo de excepciones** (`doThrow`)
    - **Retornos múltiples** (`thenReturn`)
    - **Orden de ejecución** (`InOrder`)
    - **Spies para pruebas combinadas** (`@Spy`)

Actividad

Verificación de Interacciones Simples con Mockito

Completa el archivo de prueba `GestorPedidosTest.java` rellenando las líneas comentadas. Tu objetivo es:

1. Declarar e inicializar un mock de `ServicioNotificaciones`.
2. Inyectar este mock en una instancia de `GestorPedidos`.
3. Invocar el método `procesarPedido` en la clase bajo prueba.
4. Verificar que el método `enviarNotificacion` del mock fue llamado exactamente una vez, con los argumentos correctos (`clienteEmail` y el mensaje esperado).

   Asegúrate de que tu test compile y pase correctamente.

   A continuación, se proporcionan los archivos Java necesarios:

ServicioNotificaciones.java

```
package com.ejercicio.mockito;

public class ServicioNotificaciones {
    public void enviarNotificacion(String destinatario, String mensaje) {
        System.out.println("Enviando notificación a " + destinatario + ": " + mensaje);
    }
}
```

GestorPedidos.java

```
package com.ejercicio.mockito;

public class GestorPedidos {
    private ServicioNotificaciones servicioNotificaciones;

    public GestorPedidos(ServicioNotificaciones servicioNotificaciones) {
        this.servicioNotificaciones = servicioNotificaciones;
    }

    public void procesarPedido(String idPedido, String clienteEmail) {
        System.out.println("Procesando pedido: " + idPedido + " para " + clienteEmail);
        servicioNotificaciones.enviarNotificacion(clienteEmail, "Su pedido " + idPedido
    }
}
```

GestorPedidosTest.java

```java
package com.ejercicio.mockito;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import static org.mockito.Mockito.*;

public class GestorPedidosTest {

    // 1. Declara una variable para el mock de ServicioNotificaciones
    // private ServicioNotificaciones mockServicioNotificaciones;

    // 2. Declara una variable para la instancia de GestorPedidos
    // private GestorPedidos gestorPedidos;

    @BeforeEach
    void setUp() {
        // 3. Inicializa el mock antes de cada test
        // mockServicioNotificaciones = mock(ServicioNotificaciones.class);

        // 4. Inicializa GestorPedidos inyectando el mock
        // gestorPedidos = new GestorPedidos(mockServicioNotificaciones);
    }

    @Test
    void testProcesarPedidoEnvioNotificacion() {
        String idPedido = "PED001";
        String clienteEmail = "cliente@example.com";
        String mensajeEsperado = "Su pedido PED001 ha sido procesado.";

        // 5. Llama al método que quieres probar en GestorPedidos
        // gestorPedidos.procesarPedido(idPedido, clienteEmail);

        // 6. Verifica que el método 'enviarNotificacion' del mock fue llamado
        //    una vez, con los argumentos 'clienteEmail' y 'mensajeEsperado'.
        // verify(mockServicioNotificaciones, times(1)).enviarNotificacion(eq(clienteEm
    }
}
```