Testing con JUnit

Introducción

JUnit es un marco de pruebas unitarias para el lenguaje de programación Java. Se utiliza para crear y ejecutar pruebas automáticas, lo que permite a los desarrolladores comprobar que su código funciona correctamente. El concepto central detrás de JUnit es probar pequeñas unidades de código (normalmente métodos), de forma aislada, para verificar su correcto funcionamiento.

Compatibilidad con JDK

JUnit 4

- **Compatibilidad**: JUnit 4 es compatible con versiones de Java desde **Java 5** en adelante. Esto significa que puedes usar JUnit 4 incluso si tienes un proyecto que se ejecute en versiones antiguas de Java (5, 6, 7, 8, etc.).

JUnit 5 (JUnit Jupiter)

- **Compatibilidad**: JUnit 5 requiere un **mínimo de Java 8** para funcionar, pero es compatible con versiones más recientes de Java (9, 10, 11, 17, etc.). Si estás utilizando Java 8 o superior, puedes aprovechar las nuevas características que JUnit 5 ofrece.
- **Características avanzadas**: JUnit 5 aprovecha características modernas de Java como **expresiones lambda**, **Streams**, **API de fechas**, y otros avances introducidos en Java 8 y versiones posteriores.
- **Módulos**: JUnit 5 está compuesto de diferentes módulos, entre los que destacan JUnit Jupiter (el nuevo API) y JUnit Vintage (para mantener la compatibilidad con pruebas escritas en JUnit 4).

¿Qué es una prueba unitaria?

Una prueba unitaria se enfoca en comprobar el comportamiento de una unidad individual de código, como un método o una función. El objetivo es asegurar que, dada una entrada específica, el método produce la salida esperada o se comporta de manera esperada.

Ventajas del uso de JUnit:

- **Automatización**: Permite ejecutar las pruebas de forma automática.
- **Aislamiento**: Las pruebas unitarias verifican cada parte del código por separado.
- **Facilidad de mantenimiento**: Al detectar errores temprano, ayuda a mantener el código.
- **Documentación**: Las pruebas sirven como documentación sobre cómo debería comportarse el código.

Configuración de JUnit en un Proyecto

1. **Incluir JUnit como dependencia**: JUnit suele ser incluido mediante Maven o Gradle. Para un proyecto con Maven, el siguiente código en el archivo pom.xml incluye JUnit 5:

```xml
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.7.0</version>
    <scope>test</scope>
</dependency>
```

Importar JUnit en tu clase de pruebas:

```java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
```

Estructura de una Clase de Prueba

Una clase de prueba en JUnit está compuesta por varios métodos, cada uno de los cuales prueba una unidad del código. Cada método de prueba debe:

- Tener la anotación @Test.
- Realizar aserciones (asserts) para verificar que los resultados son los esperados.

Ejemplo de Clase para Probar

Supongamos que tenemos la siguiente clase Calculadora, que queremos probar:

```java
public class Calculadora {
    public int sumar(int a, int b) {
        return a + b;
    }

    public int dividir(int a, int b) {
        if (b == 0) {
            throw new IllegalArgumentException("Divisor no puede ser
cero");
        }
        return a / b;
    }
}
```

Crear una Prueba con JUnit

1. **Probar el método sumar()**:

```java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class CalculadoraTest {

    @Test
    public void testSumar() {
        Calculadora calculadora = new Calculadora();
        int resultado = calculadora.sumar(2, 3);
        assertEquals(5, resultado);
    }
}
```

Explicación:

- Se crea una instancia de Calculadora.
- Se invoca el método sumar() y se verifica que el resultado sea igual a 5 utilizando assertEquals.

2. **Probar el método dividir() con entrada válida**:

```java
@Test
public void testDividir() {
    Calculadora calculadora = new Calculadora();
    int resultado = calculadora.dividir(10, 2);
    assertEquals(5, resultado);
}
```

Explicación: Esta prueba verifica que al dividir 10 entre 2, el resultado sea 5.

3. **Probar el método dividir() con divisor igual a cero**:

```java
@Test
public void testDividirPorCero() {
    Calculadora calculadora = new Calculadora();
    Exception exception =
assertThrows(IllegalArgumentException.class, () -> {
        calculadora.dividir(10, 0);
    });
    assertEquals("Divisor no puede ser cero",
exception.getMessage());
}
```

Explicación: En este caso, usamos assertThrows para asegurarnos de que el método lanza una excepción cuando se intenta dividir entre cero.

Tipos de Aserciones en JUnit

- **assertEquals(expected, actual)**: Verifica si el valor real es igual al esperado.
- **assertTrue(condition)**: Verifica que una condición sea verdadera.
- **assertFalse(condition)**: Verifica que una condición sea falsa.
- **assertThrows(expectedType, executable)**: Verifica que se lance una excepción de un tipo específico.
- **assertNull(object)**: Verifica que un objeto sea nulo.
- **assertNotNull(object)**: Verifica que un objeto no sea nulo.

Ciclo de Vida de las Pruebas

JUnit también ofrece la posibilidad de definir métodos que se ejecutan antes o después de todas las pruebas o antes o después de cada prueba.

- **@BeforeAll**: Se ejecuta una vez antes de todas las pruebas.
- **@AfterAll**: Se ejecuta una vez después de todas las pruebas.
- **@BeforeEach**: Se ejecuta antes de cada prueba.
- **@AfterEach**: Se ejecuta después de cada prueba.

Ejemplo:

```
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.Test;

public class CalculadoraTest {

    private Calculadora calculadora;

    @BeforeEach
    public void setUp() {
        calculadora = new Calculadora();  // Se ejecuta antes de cada
prueba
    }

    @AfterEach
    public void tearDown() {
        calculadora = null;  // Se ejecuta después de cada prueba
    }

    @Test
    public void testSumar() {
        int resultado = calculadora.sumar(2, 3);
        assertEquals(5, resultado);
    }

    // Otras pruebas...
```

```
}
```

Buenas Prácticas en Pruebas Unitarias

1. **Pruebas simples y aisladas**: Cada prueba debe verificar solo una cosa.
2. **Pruebas rápidas**: Las pruebas unitarias deben ser rápidas para facilitar su ejecución frecuente.
3. **Datos predecibles**: Usa entradas y salidas conocidas para facilitar la comprobación de resultados.
4. **Nombres descriptivos**: Los métodos de prueba deben tener nombres que describan claramente lo que están probando.
5. **Evitar dependencias externas**: No interactuar con sistemas externos como bases de datos o servicios web en pruebas unitarias.

Conclusión

JUnit es una herramienta poderosa para realizar pruebas unitarias en Java. Permite a los desarrolladores garantizar que su código funcione correctamente a lo largo del tiempo. La automatización de las pruebas mejora la calidad del software y facilita la detección de errores a medida que el proyecto crece.

Ejercicios Propuestos

1. Implementa una clase CuentaBancaria con métodos para depositar y retirar dinero. Escribe pruebas para verificar que no se pueda retirar más dinero del disponible.
2. Crea una clase ConversorTemperatura que convierta de Celsius a Fahrenheit. Implementa pruebas para verificar que las conversiones son correctas.
3. Escribe una prueba que valide que un método lanza una excepción si se le pasa un valor nulo.

Desarrollo Dirigido por Pruebas (TDD - Test Driven Development)

El **Desarrollo Dirigido por Pruebas (TDD)** es una metodología de desarrollo de software donde las pruebas unitarias se escriben antes de escribir el código funcional. El enfoque principal es garantizar que el código escrito cumpla con los requisitos desde el principio y facilite la detección temprana de errores.

Ciclo del TDD

El proceso TDD sigue un ciclo simple de tres pasos:

1. **Escribir una prueba (Red)**:
   - Antes de escribir cualquier código funcional, se escribe una prueba que cubra un pequeño fragmento de funcionalidad.
   - La prueba al principio debe fallar, porque no existe código funcional que la satisfaga.
2. **Escribir el código mínimo (Green)**:
   - Después de que la prueba falle, se escribe el código mínimo necesario para que la prueba pase. Este código no tiene que ser el más eficiente o limpio, pero debe ser suficiente para hacer que la prueba sea exitosa.
3. **Refactorizar (Refactor)**:
   - Una vez que la prueba pase, se mejora el código para hacerlo más claro, eficiente o conforme a buenas prácticas, sin cambiar su comportamiento.
   - El objetivo es mejorar la calidad del código mientras se mantienen todas las pruebas exitosas.

Este ciclo se repite para cada pequeña funcionalidad que se desee implementar.

Ventajas del TDD

1. **Mejora de la calidad del código**: Al escribir pruebas primero, los desarrolladores se ven obligados a pensar en los requisitos del sistema y en posibles errores antes de implementar la funcionalidad, lo que resulta en un código más robusto.
2. **Documentación automática**: Las pruebas sirven como documentación viva del sistema, ya que describen el comportamiento esperado del código.
3. **Facilita el refactorizado**: Dado que siempre hay pruebas que validan el comportamiento del código, los desarrolladores pueden refactorizar sin miedo a romper funcionalidades existentes.
4. **Detección temprana de errores**: Los errores se detectan a medida que se escribe el código, lo que reduce el costo de arreglarlos más adelante.

Ejemplo Simple de TDD

Supongamos que necesitas desarrollar una función para sumar dos números.

1. **Escribir la prueba**:

```java
@Test
public void testSuma() {
    Calculadora calculadora = new Calculadora();
    assertEquals(5, calculadora.sumar(2, 3));
}
```

2. **Hacer que la prueba falle**:

- En este punto, la clase Calculadora no está implementada, por lo que la prueba fallará.

3. **Escribir el código mínimo para que la prueba pase**:

```java
public class Calculadora {
    public int sumar(int a, int b) {
        return a + b;
    }
}
```

   - Ahora, la prueba pasará porque el método sumar devuelve la suma correcta.

4. **Refactorizar**:
   1. Si el código está limpio y funciona bien, no hay necesidad de refactorizar en este caso, pero es posible que en casos más complejos se requiera optimizar el código.

Desafíos del TDD

1. **Requiere disciplina**: Es tentador escribir primero el código funcional, por lo que los desarrolladores deben ser disciplinados para seguir el ciclo de pruebas.
2. **Dificultad para pruebas complejas**: Escribir pruebas para interacciones complicadas o sistemas externos (como bases de datos o APIs) puede ser más complicado y requerir el uso de **mocks** o **stubs**.
3. **No garantiza un diseño perfecto**: Aunque el TDD promueve un código más limpio y modular, no necesariamente garantiza un diseño ideal sin una buena comprensión de los principios de diseño de software.

Conclusión

El **TDD** es una metodología poderosa para desarrollar software más confiable, con menos errores y un mejor diseño, ya que fomenta la escritura de pruebas desde el principio. Aunque tiene una curva de aprendizaje y puede llevar más tiempo al principio, a largo plazo ahorra esfuerzo al reducir la cantidad de errores y refactorizaciones innecesarias.