Material descargable

Testing unitario: JUnit

egg

Pruebas unitarias

Las **pruebas unitarias** (o "testing unitario") desempeñan un papel fundamental en la validación del software al verificar el funcionamiento preciso de los componentes individuales.

En estas pruebas, se examina la funcionalidad de métodos o clases de manera aislada, sin tener en cuenta sus dependencias. De esta manera, se garantiza que cada componente trabaje correctamente y cumpla con sus objetivos específicos.

JUnit: Un marco de pruebas para Java

JUnit es un framework (marco de trabajo) popular para realizar pruebas unitarias en Java. Es simple, eficaz y ampliamente utilizado en la comunidad de desarrolladores Java.

Con JUnit, **puedes fácilmente escribir pruebas que verifiquen el comportamiento de tu código** y luego ejecutar esas pruebas para obtener informes detallados de los resultados.

Estructura de una clase Test con JUnit

Al utilizar JUnit para pruebas unitarias, es esencial importar las clases y anotaciones pertinentes. A continuación, presentamos un ejemplo de cómo estructurar una clase de prueba en JUnit:

```java
package test;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;

import src.Ejercicio1;

public class Ejercicio1Test {
    @Test
    public void testMetodo() {
        Integer resultado = Ejercicio1.metodo(2);
        Assertions.assertEquals(4, resultado);
    }
}
```

En este ejemplo, la clase de prueba se asemeja a una clase Java normal y no contiene un método main. Observamos el uso de la anotación @Test, que indica a JUnit que el método es una prueba unitaria y debe ser ejecutado como tal por el framework.

En la prueba unitaria, se invoca el método metodo() de la clase Ejercicio1 que se desea probar. Luego, mediante la clase Assertions, se verifica si el resultado es igual al valor esperado.

Si el método assertEquals devuelve true, JUnit informará a la extensión de VS Code que la prueba se completó correctamente. En caso de que assertEquals devuelva false, JUnit indicará a la extensión de VS Code que la prueba ha fallado y proporcionará un mensaje detallado que muestra la diferencia entre el resultado esperado y el resultado actual.

Convenciones de Nomenclaturas

Las convenciones de nomenclatura son pautas o reglas establecidas para nombrar elementos en el desarrollo de software, como variables, funciones, clases, archivos, etc. Estas convenciones ayudan a mejorar la legibilidad del código, facilitan la comprensión y colaboración entre programadores, y promueven una estructura coherente en el proyecto.

Convención de nomenclatura para clases

Convención de Nomenclatura para Clases	Detalles
La convención es agregar "Test" al final del nombre de la clase principal. Esta convención establece una relación clara entre la clase que está siendo probada y la clase de prueba, lo que facilita la búsqueda de las pruebas asociadas a una clase específica.	Ejemplo: "Calculadora" se convierte en "CalculadoraTest".

```
package test;
public class CalculadoraTest {
    // pruebas unitarias para Calculadora
}
```

Convención de nomenclatura para métodos

Convención de Nomenclatura para Métodos	Detalles
testMethod Name:	- Convención antigua, menos descriptiva. - Comienza con "test" seguido del nombre de la funcionalidad probada.

```
@Test
void testSuma() {
// código de prueba aquí...
}
```

should_MethodName_ExpectedBehavior_GivenCondition	- Descriptiva y clara. - Formato: should-when-given, describe comportamiento esperado con condición.

```
@Test
void should_Suma_ReturnCorrectSum_GivenMultipleNumberPairs() {
// código de prueba aquí...
}
```

MethodName_GivenCondition_ExpectedBehavior:	- Coloca la condición antes del comportamiento esperado. - Lee como una sentencia de causa y efecto.

```java
@Test
void suma_GivenMultipleNumberPairs_ReturnsCorrectSum() {
    // código de prueba aquí...
}
```

given_Precondition_When_StateUnderTest_Then_ExpectedBehavior:	- Sigue el enfoque Given-When-Then. - Muy descriptiva, pero puede resultar en nombres largos.

```java
@Test
void given_MultipleNumberPairs_When_SumaCalled_Then_ReturnsCorrectSum() {
// código de prueba aquí...
}
```

MethodName_Scenario_ExpectedResult:	- Útil para pruebas con diferentes escenarios. - Proporciona un formato claro destacando lo que se prueba y se espera.

```java
@Test
void suma_MultipleNumberPairs_CorrectSum() {
    // código de prueba aquí...
}
```

DisplayName

Desde JUnit 5, puedes usar la anotación **@DisplayName** para asignar un nombre más descriptivo y en formato de texto libre a tu prueba. Esto puede ser útil para describir la intención de la prueba de una manera más legible y además reemplaza el nombre del método en la publicación del resultado.

```java
@Test
@DisplayName("Test del método suma() con múltiples pares de números: Debería retornar la suma correcta")
void testSuma() {
    // código de prueba aquí...
}
```

Aserciones

Las aserciones, también conocidas como afirmaciones, son declaraciones que verifican si cierta condición es verdadera. Son esenciales en las pruebas

unitarias, ya que nos permiten garantizar que nuestro código funcione correctamente.

JUnit 5 proporciona la clase org.junit.jupiter.api.Assertions, que incluye una variedad de métodos estáticos para realizar diferentes tipos de afirmaciones:

Método	Descripción	Ejemplo de Uso

`assertEquals(expected, actual)`	**Verifica si dos valores son iguales. Si no lo son, la prueba fallará.**	`@Test`
		`void testSuma() {`
		`  // La suma debería ser 5`
		`  assertEquals(5, 2 + 3);`
		`}`

`assertNotEquals(expected, actual)`	**Verifica si dos valores NO son iguales. Si lo son, la prueba fallará.**	`@Test`
		`void testSuma() {`
		`  // La suma no debería ser 6`
		`  assertNotEquals(6, 2 + 3);`
		`}`

`assertTrue(condition)`	**Verifica si una condición es verdadera. Si no lo es, la prueba fallará.**	`@Test`
		`void testIsEven() {`
		`  // 4 debería ser par`
		`  assertTrue(4 % 2 == 0);`
		`}`

`assertFalse(condition)`	**Verifica si una condición es falsa. Si no lo es, la prueba fallará.**	`@Test`
		`void testIsOdd() {`
		`  // 4 no debería ser impar`
		`  assertFalse(4 % 2 != 0);`
		`}`

`assertNull(value)`	**Verifica si un valor es nulo. Si no lo es, la prueba fallará.**	`@Test`
		`void testNullValue() {`
		`  String str = null;`
		`  // La variable debería ser nula`
		`  assertNull(str);`
		`}`

`assertNotNull(value)`	**Verifica si un valor NO es nulo. Si lo es, la prueba fallará.**	`@Test`
		`void testNotNullValue() {`
		`  String str = "Hola mundo";`
		`  // La variable no debería ser nula`
		`  assertNotNull(str);`
		`}`

`assertSame(expected, actual)`	**Verifica si dos referencias de objetos apuntan al mismo objeto. Si no lo hacen, la prueba fallará.**	`@Test`
		`void testSameObject() {`
		`  String str1 = "Hola mundo";`
		`  String str2 = str1;`
		`  // Las variables deberían referenciar al mismo objeto`
		`  assertSame(str1, str2);`
		`}`

`assertNotSame(expected, actual)`	**Verifica si dos referencias de objetos NO apuntan al mismo objeto. Si lo hacen, la prueba fallará.**	`@Test`
		`void testNotSameObject() {`
		`  String str1 = new String("Hola mundo");`
		`  String str2 = new String("Hola mundo");`
		`  // Las variables no deberían referenciar al mismo objeto`
		`  assertNotSame(str1, str2);`
		`}`

`assertArrayEquals(expectedArray, actualArray)`	**Verifica si dos arrays son iguales. Si no lo son, la prueba fallará.**	`@Test`
		`void testArrayEquality() {`
		`  int[] array1 = {1, 2, 3};`
		`  int[] array2 = {1, 2, 3};`
		`  // Los arrays deberían ser iguales`
		`  assertArrayEquals(array1, array2);`
		`}`

`assertThrows(expectedType, executable)`	**Verifica si una operación lanza una excepción del tipo esperado.**	`@Test`
		`void testException() {`
		`  // Debería lanzar ArithmeticException`
		`  assertThrows(ArithmeticException.class, () -> {`
		`  int division = 5 / 0;`
		`  });`
		`}`

`assertEquals(double expected, double actual, double delta)`	**Compara números de punto flotante permitiendo una diferencia de precisión.**	`@Test`
		`public void testSquareRoot() {`
		`  // El valor de la raíz cuadrada de 4 debería ser 2`
		`  assertEquals(2.0, Math.sqrt(4.0));`
		`  // La raíz cuadrada de 2 debería ser cercana a 1.4142`
		`  assertEquals(1.4142, Math.sqrt(2.0), 0.0001);`
		`}`

Mensaje personalizado de error :

Todos los métodos mencionados anteriormente también **tienen una sobrecarga que acepta un parámetro adicional de tipo String**. Este parámetro nos permite agregar un mensaje personalizado que se imprimirá en caso de que la afirmación falle. El mensaje personalizado puede ser útil para proporcionar contexto y comprender por qué la prueba ha fallado.

```
@Test
void testSuma() {
    assertEquals(5, 2 + 3, "La suma debería ser 5");}
```

Estrategia triple A (Arrange, Act, Assert)

La estrategia AAA es un patrón comúnmente utilizado para organizar y escribir pruebas unitarias, dividiendo el proceso en tres fases principales:

- **Arrange (Organizar):** En esta fase, se configura el entorno de prueba. Se crean objetos e instancias necesarios, se establecen estados iniciales y, opcionalmente, se configuran objetos simulados o mocks.
- **Act (Actuar):** Durante esta fase, se invoca el código que se está probando. Generalmente, implica llamar a un método y proporcionar los parámetros necesarios.
- **Assert (Afirmar):** En esta fase, se verifica si la acción produjo el resultado esperado. Se utilizan afirmaciones (assertions) para realizar esta verificación. Las afirmaciones son declaraciones que lanzarán una excepción si la condición especificada no se cumple.

A continuación, te mostramos un ejemplo de cómo se aplica la estrategia AAA en una prueba:

```
public class CalculadoraTest {
    @Test
    void testSuma() {
        // Arrange
        int numero1 = 4;
        int numero2 = 5;
        // Act
        int resultado = Calculadora.suma(numero1, numero2);
        // Assert
        assertEquals(9, resultado, "La suma de 4 y 5 debería ser 9"); }
}
```

En el ejemplo, se puede observar lo siguiente:

- **Arrange:** Se definen dos números que serán sumados.
- **Act:** Se llama al método estático *suma()* de la clase Calculadora, pasando los dos números como argumentos.
- **Assert:** Se verifica que el resultado de la suma sea igual a 9. En caso contrario, la prueba fallará y se mostrará el mensaje *"La suma de 4 y 5 debería ser 9"*.

Esta estructura de pruebas ayuda a que sean más legibles y comprensibles, ya que se puede identificar claramente la configuración utilizada, la acción probada y los resultados esperados.

Ciclo de vida de las pruebas unitarias

El ciclo de vida de las pruebas unitarias se refiere al proceso y los pasos que JUnit sigue cuando se ejecutan los tests de una clase. Estos pasos pueden ser personalizados mediante el uso de anotaciones específicas:

Anotación	Descripción	Ejemplo de Uso

`@BeforeAll`	**Método estático que se ejecuta una vez antes de todos los métodos de prueba en la clase de prueba.**	`class MyTestClass { @BeforeAll static void initAll() { // Código para configurar el estado antes de todas las pruebas ... } }`
`@BeforeEach`	**Método no estático que se ejecuta antes de cada método de prueba individual en la clase de prueba.**	`class MyTestClass { @BeforeEach void setUp() { // Código para configurar antes de cada prueba ... } }`
`@Test`	**Anotación que se aplica a cada método de prueba individual.**	`class MyTestClass { @Test void myTest() { // Código de la prueba ... } }`
`@AfterEach`	**Método no estático que se ejecuta después de cada método de prueba individual en la clase de prueba.**	`class MyTestClass { @AfterEach void tearDown() { // Código para limpiar el estado después de cada prueba... } }`
`@AfterAll`	**Método estático que se ejecuta una vez después de todos los métodos de prueba en la clase de prueba.**	`class MyTestClass { @AfterAll static void tearDownAll() { // Código para limpiar el estado después de todas las pruebas aquí } }`

Estas anotaciones proporcionan un gran control sobre el ciclo de vida de las pruebas, permitiendo configurar y limpiar el estado para pruebas individuales o para todas las pruebas. Cuando identifiques código repetitivo en tus pruebas, es muy probable que puedas mejorar su legibilidad y mantenibilidad al extraer ese código repetitivo a un método y utilizar una de las anotaciones del ciclo de vida.