Guía Laboratorio de Ingeniería del Software 2

Guía 01 - Pruebas Unitarias

Prof. Hernan Nina Hanco

OBJETIVO:

- Conocer el desarrollo de pruebas unitarias utilizando el framework Junit4 en Java.

TRABAJO PREPARATORIO:

- Framework Junit4: JUnit 4 es un popular marco de prueba unitaria (testing framework) para el lenguaje de programación Java. Se utiliza ampliamente en el desarrollo de software para automatizar y simplificar la tarea de escribir y ejecutar pruebas unitarias en aplicaciones Java. JUnit 4 proporciona una estructura y un conjunto de anotaciones que permiten a los desarrolladores crear pruebas unitarias de manera efectiva y organizada.

DESARROLLO DE LA GUÍA:

Paso 1: Crear un nuevo proyecto

1. Abre NetBeans 8.0.
2. Ve a "File" -> "New Project".
3. Selecciona "Java" en la categoría "Categories".
4. Elije "Java Application" y haz clic en "Next".
5. Da un nombre al proyecto (por ejemplo, "CalculadoraApp") y especifica la ubicación donde deseas guardarlo. Luego, haz clic en "Finish".

Paso 2: Crear la clase Calculadora.java

1. En el proyecto recién creado, en la carpeta "Source Packages", haz clic derecho, selecciona "New" y luego "Java Class".
2. Dale el nombre “Calculadora” y asegúrate de que esté en el mismo paquete que tu proyecto (por defecto, es el paquete `calculadoraapp`).
3. Escribe el código de la clase “Calculadora” dentro del archivo “Calculadora.java” que creaste.

Aquí está la clase “Calculadora” para referencia:

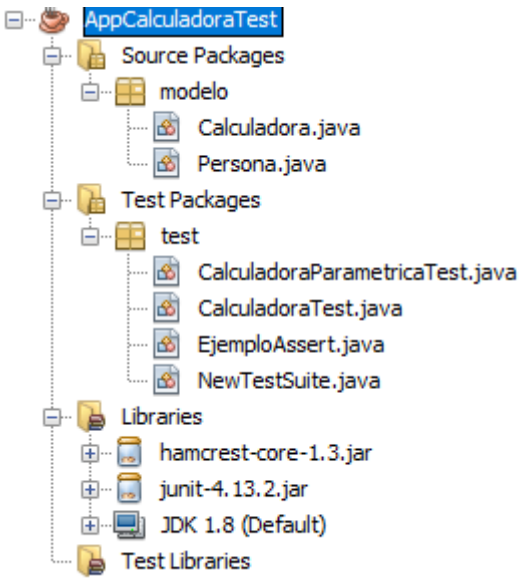
```
package calculadoraapp;
public class Calculadora {
    private int ans;
    public Calculadora() {
        this.ans = 0;
    }
    public int add(int a, int b) {
        this.ans = a + b;
        return this.ans;
    }
    public int add(int v) {
        this.ans += v;
        return this.ans;
    }
    public int sub(int a, int b) {
```

```
        this.ans = a - b;
        return this.ans;
    }
    public int sub(int v) {
        this.ans -= v;
        return this.ans;
    }
    public int ans() {
        return this.ans;
    }
    public void clear() {
        this.ans = 0;
    }
}
```

Paso 3: Agregar las bibliotecas de JUnit y Hamcrest a tu proyecto en NetBeans 8.0

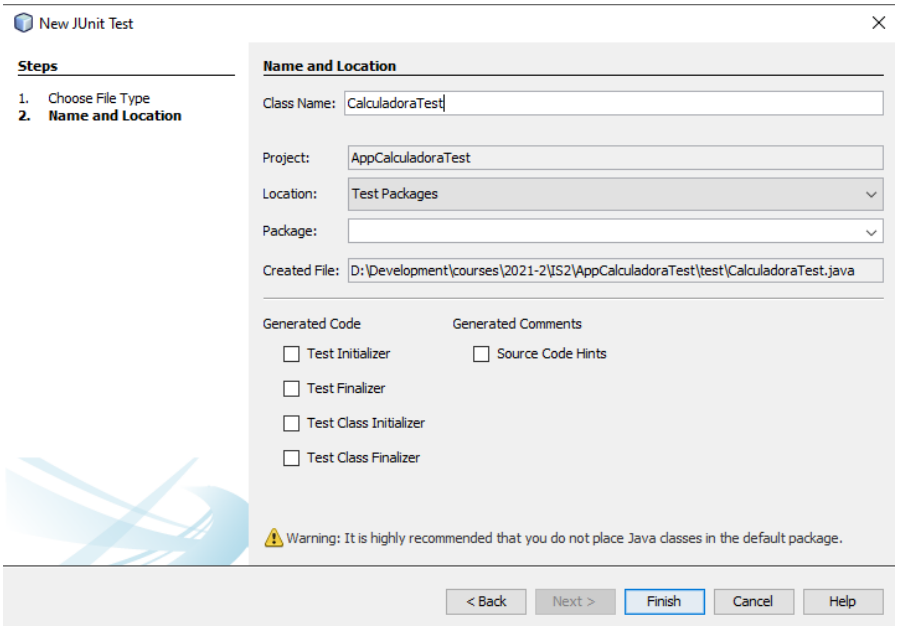
En NetBeans, abre tu proyecto "CalculadoraApp".

1. Haz clic derecho en la carpeta "Libraries" en el proyecto en el panel "Projects".
2. Selecciona "Add JAR/Folder".
3. En el cuadro de diálogo " Add JAR/Folder ", ubícate donde tienes descargado las librerías de Junit y selecciona el archivo Junit-xx.jar y hamcrest-core-xxx.jar".proyecto.



Paso 4: Crear la clase de prueba CalculadoraTest.java

1. En el proyecto, en la carpeta "Test Packages", haz clic derecho, selecciona "New" y luego "JUnit Test Case".
2. En el cuadro de diálogo "New JUnit Test Case", escribe como nombre de clase CalculadoraTest.
3. Haz clic en "Finish".



Paso 5: Escribir pruebas unitarias

Dentro de la clase `CalculadoraTest`, puedes escribir las pruebas unitarias siguiendo los ejemplos:

```
package calculadoraapp;
import org.junit.Before;
import org.junit.After;
import org.junit.Test;
import static org.junit.Assert.*;

public class CalculadoraTest {
    private Calculadora cal;
    @Before
    public void before() {
        System.out.println("Before()");
        cal = new Calculadora();
    }

    @After
    public void after() {
        System.out.println("After()");
        cal.clear();
    }

    @Test
    public void sumaTest() {
        System.out.println("Suma()");
        int resultado = cal.add(2, 3);
        int esperado = 5;
        assertEquals(esperado, resultado);
    }

    @Test
    public void sumaTest1() {
        System.out.println("Suma()");
        int resultado = cal.add(-2, -3);
```

```
        int esperado = -5;
        assertEquals(esperado, resultado);
    }

    @Test
    public void sumaTest2() {
        System.out.println("Suma()");
        int resultado = cal.add(0, 0);
        int esperado = 0;
        assertEquals(esperado, resultado);
    }

    @Test
    public void restaTest() {
        System.out.println("Resta()");
        int resultado = cal.sub(3, 5);
        int esperado = -2;
        assertEquals(esperado, resultado);
    }
}
```

Paso 6: Ejecutar las pruebas

Para ejecutar las pruebas unitarias en NetBeans 8.0, haz clic derecho en la clase “CalculadoraTest” en la vista "Projects" y selecciona "Test File". Observa la salida en la consola de NetBeans para verificar si todas las pruebas pasaron o si alguna falló.

Paso 7: Interpretar los resultados

Si todas las pruebas pasaron, verás un mensaje de éxito en la consola de NetBeans. Si alguna prueba falló, NetBeans te proporcionará información detallada sobre qué prueba falló y qué valor esperabas frente al valor real.

EJERCICIOS PROPUESTOS

Ejercicio 1: Pruebas de suma y resta

Escribe pruebas unitarias utilizando JUnit 4 para los siguientes escenarios:

- Prueba la suma de números positivos.
- Prueba la suma de números negativos.
- Prueba la suma de ceros.
- Prueba la resta de números positivos.
- Prueba la resta de números negativos.
- Prueba la resta de ceros.

Asegúrate de que todas las pruebas pasen correctamente y que estén organizadas en una clase de prueba separada.

Ejercicio 2: Pruebas de funciones adicionales

Extiende la clase Calculadora con funciones adicionales (por ejemplo, multiplicación y división) y escribe pruebas unitarias para estas funciones. Asegúrate de cubrir varios casos, incluyendo divisiones por cero, números negativos y positivos.

Ejercicio 3: Pruebas de excepciones

Modifica la clase Calculadora para que genere excepciones en casos como la división por cero o la raíz cuadrada de números negativos. Luego, escribe pruebas unitarias para verificar que las excepciones se generen correctamente en estos casos.

Ejercicio 4: Pruebas con Timeout para la clase Calculadora

Si deseas simular una operación que demore en una prueba unitaria con JUnit 4, puedes utilizar un enfoque conocido como "prueba de tiempo". Esto implica agregar un retardo o pausa deliberada en el código de tu prueba para simular una operación que toma tiempo. Puedes hacerlo utilizando la clase Thread.sleep() para pausar la ejecución de la prueba durante un período de tiempo especificado. Aquí tienes un ejemplo de cómo hacerlo:

Supongamos que tienes una clase Calculadora y deseas simular una operación de suma que toma tiempo:

```
package calculadoraapp;
public class Calculadora {
    public int sumaLenta(int a, int b) {
        // Simula una operación lenta
        try {
            Thread.sleep(5000); // Espera durante 5 segundos
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
        return a + b;
    }
}
```

En el código anterior, el método sumaLenta realiza una pausa de 5 segundos utilizando Thread.sleep() antes de realizar la suma.

Ahora, puedes escribir una prueba que verifique que esta operación tome tiempo utilizando @Test junto con la anotación timeout:

```
package calculadoraapp;
import org.junit.Test;
import static org.junit.Assert.assertEquals;
public class CalculadoraTest {
    @Test(timeout = 6000)
// Establece un tiempo de espera de 6 segundos
    public void testSumaLenta() {
        Calculadora calculadora = new Calculadora();
        int resultado = calculadora.sumaLenta(2, 3);

        assertEquals(5, resultado); // Verifica el resultado de
la suma
    }
}
```

En el código de prueba, hemos establecido un tiempo de espera de 6 segundos con timeout = 6000. Como la operación sumaLenta demora 5 segundos en ejecutarse, la prueba debería completarse dentro del tiempo de espera y pasar sin problemas. Si la operación demorara más de 6 segundos, la prueba fallaría debido al tiempo de espera agotado.

Este enfoque te permite simular operaciones lentas en tus pruebas unitarias y verificar que no excedan un cierto tiempo de ejecución. Ten en cuenta que esto es útil para probar comportamientos en situaciones en las que el tiempo de respuesta es importante, como operaciones de red o interacciones con servicios externos.

Ejercicio 5: Comparación de Arreglos de Números

Desarrolla una prueba que compare dos arreglos de números enteros y determine si son iguales. Este ejercicio incluirá arreglos más complejos y desafiantes. A continuación, se presentan varios ejemplos de arreglos para tu prueba:

- Arreglo 1: [1, 2, 3, 4, 5]
- Arreglo 2: [1, 2, 3, 4, 5]
- Arreglo 3: [5, 4, 3, 2, 1]
- Arreglo 4: [1, 3, 5, 7, 9]
- Arreglo 5: [1, 2, 2, 4, 5]

Tu tarea es crear una prueba que verifique si estos arreglos son iguales. Asegúrate de considerar arreglos con diferentes tamaños, orden de elementos variados y valores distintos. La prueba debe determinar con precisión si cada par de arreglos es igual o diferente.

Ten en cuenta que dos arreglos son iguales si tienen la misma longitud y los mismos elementos en el mismo orden. Tu prueba debe abordar esta definición de igualdad y aplicarla a los ejemplos proporcionados, demostrando si los arreglos son iguales o no.

Ejercicio 6: Comparación de Números Decimales con Tolerancia

Desarrolla una prueba que verifique si la suma de dos números decimales es igual a 0.3 con una tolerancia de precisión ajustada. Este ejercicio implica trabajar con números decimales y considerar una tolerancia específica para determinar la igualdad. A continuación, se presentan ejemplos de sumas para tu prueba:

1. Suma 1: 0.1 + 0.2
2. Suma 2: 0.02 + 0.03
3. Suma 3: 0.12345 + 0.45678
4. Suma 4: 0.987 + 0.002

Tu tarea consiste en crear una prueba que verifique si estas sumas de números decimales son iguales a 0.3 dentro de una tolerancia específica. Esto significa que la prueba debe permitir pequeñas diferencias entre el resultado real de la suma y el valor objetivo de 0.3, siempre y cuando esa diferencia esté dentro de la tolerancia especificada.

Por ejemplo, si la tolerancia es de 0.001, la prueba considerará que las sumas son iguales si la diferencia entre el resultado real y 0.3 es menor o igual a 0.001. Debes diseñar la prueba de manera que sea capaz de manejar estas complejidades y determinar con precisión si las sumas son iguales dentro de la tolerancia definida.

Ejercicio 6: Comparación de Objetos Personalizados

Escribe una prueba que evalúe si dos objetos de la clase MiObjeto con valores específicos cumplen con la condición de igualdad. Este ejercicio implica trabajar con objetos personalizados y determinar si son iguales en función de ciertos criterios. A continuación, se presentan ejemplos con objetos MiObjeto para tu prueba:

Ejemplo 1:

```
MiObjeto objeto1 = new MiObjeto("ABC", 42);

MiObjeto objeto2 = new MiObjeto("ABC", 42);
```

Ejemplo 2:

```
MiObjeto objeto3 = new MiObjeto("XYZ", 100);

MiObjeto objeto4 = new MiObjeto("ABC", 42);
```

Tu tarea es crear una prueba que verifique si estos objetos MiObjeto son iguales según ciertos criterios que determines. Puedes considerar la igualdad basada en propiedades específicas de los objetos, como el valor de una cadena y un número en este caso.

Por ejemplo, en el primer ejemplo, podrías diseñar la prueba para que los objetos objeto1 y objeto2 se consideren iguales si sus cadenas y números coinciden. En el segundo ejemplo, podrías diseñar la prueba para que los objetos objeto3 y objeto4 se consideren diferentes debido a diferencias en sus propiedades.

La prueba debe ser capaz de manejar estos casos y determinar con precisión si los objetos MiObjeto son iguales según los criterios definidos.

Ejercicio 7: Pruebas de lógica más compleja para una clase CuentaBancaria

1. Crea un nuevo proyecto en NetBeans llamado "CuentaBancariaTests".
2. Implementa una clase llamada CuentaBancaria que tenga los siguientes métodos básicos:
    a. saldoInicial() que establezca el saldo inicial de la cuenta.
    b. depositar(double monto) que permita depositar dinero en la cuenta.
    c. retirar(double monto) que permita retirar dinero de la cuenta.
    d. obtenerSaldo() que devuelva el saldo actual de la cuenta.
    e. Realizar la transferencia de una cuenta a otra.
3. Escribe pruebas unitarias utilizando JUnit 4 para verificar la lógica de la clase CuentaBancaria. Algunos ejemplos de pruebas que puedes realizar incluyen:
    a. Verificar que el saldo inicial sea el correcto.
    b. Depositar una cantidad y verificar que el saldo se actualice correctamente.
    c. Intentar retirar una cantidad mayor que el saldo y verificar que la operación sea rechazada.
    d. Realizar varios depósitos y retiros y verificar que el saldo sea coherente.
    e. Puedes también agregar pruebas adicionales para casos límite, como intentar retirar una cantidad negativa o realizar operaciones con montos cero.
4. Asegúrate de que todas las pruebas pasen correctamente y que estén organizadas en una clase de prueba separada llamada CuentaBancariaTest.
5. Utiliza aserciones de JUnit, como assertEquals, para verificar que los resultados sean los esperados en cada prueba. Toma en cuenta que assertEquals para valores double debe considerarse un margen de error.