JUnit

# Introducción

Las pruebas son el proceso de comprobación de la funcionalidad de una aplicación para garantizar que se ejecuta según los requisitos. Las pruebas unitarias entran en escena en el nivel de los desarrolladores; es la prueba de una sola entidad (clase o método). Las pruebas unitarias desempeñan un papel fundamental a la hora de ayudar a una empresa de software a ofrecer productos de calidad a sus clientes.

Las pruebas unitarias pueden realizarse de dos maneras: pruebas manuales y pruebas automatizadas.

# Sobre JUnits

JUnit es un framwork de pruebas unitarias para el lenguaje de programación Java. Desempeña un papel crucial en el desarrollo basado en pruebas, y es una familia de framework de pruebas unitarias conocidos colectivamente como xUnit.

JUnit promueve la idea de "primero probar y luego codificar", que hace hincapié en la creación de los datos de prueba para un fragmento de código que puede ser probado primero y luego implementado. Este enfoque es como "probar un poco, codificar un poco, probar un poco, codificar un poco". Aumenta la productividad del programador y la estabilidad del código del programa, lo que a su vez reduce el estrés del programador y el tiempo dedicado a la depuración.

## Características de Junit

* Proporciona anotaciones para identificar los métodos de prueba.
* Proporciona afirmaciones para probar los resultados esperados.
* Proporciona ejecutores de pruebas para ejecutarlas.
* Las pruebas de JUnit permiten escribir códigos más rápidamente, lo que aumenta la calidad.
* JUnit es elegantemente simple. Es menos complejo y requiere menos tiempo.
* Las pruebas de JUnit se pueden ejecutar automáticamente y comprueban sus propios resultados y proporcionan información inmediata. No es necesario peinar manualmente un informe de resultados de pruebas.
* Las pruebas de JUnit pueden organizarse en conjuntos de pruebas que contienen casos de prueba e incluso otros conjuntos de pruebas.
* JUnit muestra el progreso de las pruebas en una barra que es verde si la prueba se está ejecutando sin problemas, y se vuelve roja cuando una prueba falla.

Un caso de prueba unitaria (Test Case) es una parte del código que garantiza que otra parte del código (método) funciona como se espera. Para lograr los resultados deseados rápidamente, se requiere un framework de pruebas. JUnit es un marco de pruebas unitarias perfecto para el lenguaje de programación Java.

Un caso de prueba unitaria escrito formalmente se caracteriza por una entrada conocida y una salida esperada, que se elabora antes de ejecutar la prueba. La entrada conocida debe probar una precondición y la salida esperada debe probar una postcondición.

Debe haber al menos dos casos de pruebas unitarias para cada requisito: una prueba positiva y otra negativa. Si un requisito tiene sub-requisitos, cada sub-requisito debe tener al menos dos casos de prueba como positivo y negativo.

# Uso del Framework

El framework de pruebas JUnit proporciona las siguientes funcionalidades importantes:

## Fixtures

El fixture es un estado fijo de un conjunto de objetos que se utiliza como línea de base para la ejecución de las pruebas. El propósito de un fixture de prueba es asegurar que existe un entorno conocido y fijo en el que se ejecutan las pruebas para que los resultados sean repetibles.

Esto es conocido como los métodos setUp(), que se ejecuta antes de cada invocación de la prueba, y tearDown(), que se ejecuta después de cada método de prueba.

## Test Suites

Un conjunto de pruebas (Test Suites) agrupa algunos casos de pruebas unitarias y los ejecuta juntos. En JUnit, las anotaciones @RunWith y @Suite se utilizan para ejecutar el conjunto de pruebas.

# Implementación

## Creación del Test Case

### Dependencia

Para utilizar Junit versión 5 en un proyecto Maven, debemos indicar las siguientes dependencia.

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter-api</artifactId>

<version>5.5.2</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter-engine</artifactId>

<version>5.5.2</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.junit.platform</groupId>

<artifactId>junit-platform-runner</artifactId>

<version>1.5.2</version>

<scope>test</scope>

</dependency>

A parte, tambien debemos configurar el plugin surefire:

<build>

<plugins>

<plugin>

<artifactId>maven-surefire-plugin</artifactId>

<version>2.22.2</version>

</plugin>

<plugin>

<artifactId>maven-failsafe-plugin</artifactId>

<version>2.22.2</version>

</plugin>

</plugins>

</build>

Es importante considerar que esta versión requiere utilizar java 8. Una vez indicada la dependencia, podemos utilizar sus diferentes anotaciones y armar nuestro proceso de prueba.

### Caso de prueba

Para indicar que métodos son los casos de prueba que deseamos ejecutar, indicamos la anotación **@Test**. Esto hará que automáticamente tanto Junit como Maven los reconozca como casos de pruebas unitarios y serán ejecutados al indicar el comando test.

@Test

public void testCase1() {

....

}

### Configurando el escenario de prueba

Como es común, muchas veces los casos de prueba necesitan escenarios comunes para poder ejecutarse. Estos escenarios pueden prepararse implementando los Fixtures mediante las siguientes anotaciones:

* @BeforeAll**y**@BeforeEach:

@BeforeEach se coloca sobre un método para configurar parte del escenario de prueba antes de que se ejecute cada caso, mientras que @BeforeAll se implementa sobre un método estático que se ejecuta antes de que se lance el primer test case de la clase. Ejemplo:

@BeforeAll

static void setup() {

log.info("@BeforeAll - se ejecuta una sola vez antes de todos los casos de prueba de esta clase.");

}

@BeforeEach

void init() {

log.info("@BeforeEach - se ejecuta antes de cada caso de prueba de esta clase.");

}

* *@AfterEach* and *@AfterAll*

@AfterEach se coloca sobre un método para configurar parte del escenario de prueba después de que se ejecuta cada caso, mientras que @AfterAll se implementa sobre un método estático que se ejecuta después de que se lance el ultimo test case de la clase. Ejemplo:

@AfterEach

void tearDown() {

log.info("@AfterEach - ejecuta despues de cada metodo de prueba.");

}

@AfterAll

static void done() {

log.info("@AfterAll - ejecuta despues del ultimo metodo de prueba.");

}

### Runner

Todos los casos de prueba necesitan tener indicado un ejecutor o runner para que el framework sepa cómo tratarlos. Por defecto se ejecutan con el runner que el IDE utilizado tiene asociado, pero en caso de necesitar personalizarlo utilizamos la anotación @RunWith a nivel de clase indicando el ejecutor que deseamos. Uno de los ejecutores recomendados es JUnitPlatform, como se ilustra en el siguiente ejemplo:

@RunWith(JUnitPlatform.class)

public class TestCaseClassTest {

...

}

## Test Suite

Una vez creados todos los casos de pruebas que deseamos comprobar, podemos agruparlos para luego ejecutarlos para comprobar funcionalidades especificas del programa.

### Runner

En la agrupación se debe indicar con que ejecutor o runner se desea utilizar para ejecutar los casos de prueba por defecto. Esto se hace utilizando la anotación @RunWith seguido de su ejecutor. Estos son algunos ejemplos de Runners:

* Suite.class

@RunWith(Suite.class)

public class TestCaseClassTest {

...

}

* JUnitPlatform.class

@RunWith(JUnitPlatform.class)

public class TestCaseClassTest {

...

}

### Agrupación de Test Cases

Para agrupar los test cases deseados, existen 3 mecanismos:

* SuiteClass: Anotación disponible desde la versión 4.4 y se utiliza para indicar el listado de clases que se desea agrupar. Ejemplo:

@RunWith(JUnitPlatform.class)

@SuiteClasses({TestClass1.class, TestClass2.class})

public class TestSuiteClass {}

* SelectClasses: Anotación disponible desde la versión 5.0 y realiza la misma función que la anotación @SuiteClass. Ejemplo:

@RunWith(JUnitPlatform.class)

@SelectClasses({TestCase1.class, TestCase2.class})

public class TestSuiteClass {}

* SelectPackages: Anotacion disponible desde la versión 5.0 y se utiliza para indicar en que paquetes se encuentra los casos de prueba a agrupar. Ejemplo:

@RunWith(JUnitPlatform.class)

@SelectPackages({"edu.fpdual.testprueba","edu.fpdual.testsimples"})

public class AllUnitTest {}

# Suposiciones

Las suposiciones (Assumptions) se utilizan para ejecutar las pruebas sólo si se cumplen ciertas condiciones. Esto se suele utilizar para condiciones externas que son necesarias para que la prueba se ejecute correctamente, pero que no están directamente relacionadas con lo que se está probando.

Puedes declarar una suposición con:

* assumeTrue()

@Test

void trueAssumption() {

assumeTrue(5 > 1);

assertEquals(5 + 2, 7);

}

* assumeFalse()

@Test

void falseAssumption() {

assumeFalse(5 < 1);

assertEquals(5 + 2, 7);

}

* assumingThat().

@Test

void assumptionThat() {

String someString = "Just a string";

assumingThat(

someString.equals("Just a string"),

() -> assertEquals(2 + 2, 4)

);

}

# Validaciones

Todas las pruebas unitarias tienen el propósito de validar un resultado, sea un conjunto de valores o un error. Esto le da el significado y el propósito de poder confirmar que el comportamiento esperado de un fragmento de código realiza lo que se espera de el dado una entrada especifica.

## Afirmaciones

Para confirmar un resultado esperado, digamos un conjunto de datos o unos valores en específico, Junit cuenta con afirmaciones (Assetions) que ayudan a poder validar/confirmar estos datos. Existen muchos métodos para realizar estas acciones, todos estáticos, y pueden ser encontrados dentro de la clase [**org.junit.jupiter.api.Assertions**](https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/Assertions.html).

Alguno de los asserts (afirmaciones) comúnmente utilizadas son:

* assertTrue y assertFalse: espera que un valor sea verdadero o falso.

@Test

public void testCase() {

assertTrue(age > 18);

}

* assertNull y assertNotNull: compara si un valor dado es nulo o no.

@Test

public void testCase() {

assertNull(respuesta);

assertNotNull(respuesta);

}

* assertEquals: compara que dos valores sean iguales.

@Test

public void testCase() {

assertEquals(BigDecimal.ONE, BigDecimal.ONE);

}

## Excepciones

Para validar que bajo ciertos parámetros un caso de prueba lanza una excepción, podemos utilizar varias opciones:

* assertThrows: valida si la ejecución de un proceso lanza una excepción en particular. Puede utilizarse de dos formas:
  + Validar si la invocación del proceso lanza directamente una excepción.

@Test

public void testCase() {

final Double value = null;

assertThrows(NullPointerException.class, () -> value.floatValue());

}

* + Capturar la excepción lanzada por un proceso dada un caso de prueba para poder validar sus datos.

@Test

public void testCase() {

Throwable exception = assertThrows(UnsupportedOperationException.class, () -> {

throw new UnsupportedOperationException("Not supported");

});

assertEquals(exception.getMessage(), "Not supported");

}

# Hamcrest

Hamcrest se utiliza comúnmente con junit y otros frameworks de pruebas para hacer aserciones. En concreto, en lugar de utilizar los numerosos métodos de aserción de junit, sólo utilizamos la única sentencia assertThat de la API con los correspondientes comparadores.

@Test

public void testCase() {

String a = "foo";

String b = "FOO";

MatcherAssert.assertThat(a, Matchers.equalToIgnoringCase(b));

}

## Dependencia

<dependency>

<groupId>org.hamcrest</groupId>

<artifactId>hamcrest-all</artifactId>

<version>1.3</version>

</dependency>

## Matchers

Este framework proporciona una serie de Matchers para diferentes propósitos, entre los que están:

* Object Matcher: prove matchers para comparer objetos de diversas formas.

@Test

public void given2Classes\_whenOneInheritsFromOther\_thenCorrect(){

assertThat(Cat.class,typeCompatibleWith(Animal.class));

}

}

* Bean Matcher: inspecciona propiedades de un objeto java.

@Test

public void givenBean\_whenHasCorrectValue\_thenCorrect() {

Person person = new Person("FP", "Accenture");

assertThat(person, hasProperty("address", equalTo("Accenture")));

}

* Collection Matcher: valida diferentes aspectos de una coleccion como si esta vacia, si contiene ciertos datos, entre otros.

@Test

public void testCase() {

List<String> hamcrestMatchers = Arrays.asList("collections", "beans", "text", "number");

assertThat(hamcrestMatchers, hasSize(4));

}

* Number Matcher: prove metodos para validar valores numéricos.

@Test

public void givenAnInteger\_whenGreaterThanOrEqTo5\_thenCorrect() {

assertThat(5, greaterThanOrEqualTo(5));

}

* Text Matcher: dispone de diversos métodos para validar textos.

@Test

public void givenString\_whenEmpty\_thenCorrect() {

String str = "";

assertThat(str, isEmptyString());

}

* Matcher personalizado: el programador puede hacer su propio matcher extendiendo de la clase “*TypeSafeMatcher” y sobreescribiendo los métodos “describeTo” para personalizar el mensaje de la validación en caso de error y “matchesSafely” para indicar como validar los datos que estamos comparando.*

Matcher personalizado:

public class IsPositiveInteger extends TypeSafeMatcher<Integer> {

public void describeTo(Description description) {

description.appendText("a positive integer");

}

@Factory

public static Matcher<Integer> isAPositiveInteger() {

return new IsPositiveInteger();

}

@Override

protected boolean matchesSafely(Integer integer) {

return integer > 0;

}

}

Uso del matcher:

@Test

public void givenInteger\_whenAPositiveValue\_thenCorrect() {

int num = 1;

assertThat(num, isAPositiveInteger());

}

## API

El núcleo de la API de Hamcrest debe ser utilizado por los proveedores de frameworks de terceros. Sin embargo, nos ofrece algunas construcciones geniales para hacer nuestras pruebas unitarias más legibles y también algunos matchers del núcleo que se pueden utilizar con la misma facilidad. Entre las muchos operaciones que nos ofrece, tenemos:

* Is: Valida si el matcher o valor utilizado utilizado retorna true.

@Test

public void given2Strings\_whenIsEqualRegardlessWhiteSpace\_thenCorrect() {

String str1 = "text";

String str2 = " text ";

assertThat(str1, is(equalToIgnoringWhiteSpace(str2)));

}

* Not: valida si el matcher o valor utilizado retorna false.

@Test

public void given2Strings\_whenNotEqual\_thenCorrect() {

String str1 = "text";

String str2 = "texts";

assertThat(str1, not(str2));

}

* anyOf: valida que alguno de los matcher indicados retorna true.

@Test

public void givenString\_whenMeetsAnyOfGivenConditions\_thenCorrect() {

String str = "calligraphy";

String start = "call";

String end = "foo";

assertThat(str, anyOf(startsWith(start), containsString(end)));

}

# Referencias

Para investigar más sobre el tema, pueden revisar los siguientes enlaces:

<https://www.baeldung.com/junit>

<https://junit.org/junit5/docs/current/user-guide/#overview>

<https://www.baeldung.com/java-junit-hamcrest-guide>