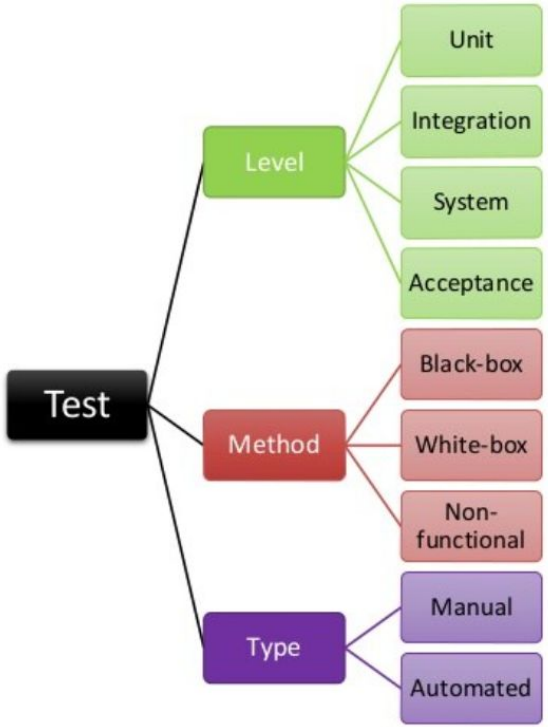
Pruebas Unitarias

UNJu - Programación Orientada a Objetos

Ing. José Zapana

Pruebas de software



- Pruebas unitarias (componentes aislados)
- Pruebas de integración (diferentes componentes)
- Pruebas de sistema (todos los componentes)
- Pruebas de aceptación (pruebas de usuario)
- Pruebas de caja negra (funcionales)
- Pruebas de caja blanca (estructurales)
- Pruebas no funcionales (rendimiento, seguridad,…)
- Pruebas manuales
- Pruebas automáticas (*frameworks* de prueba)

¿Qué son las Pruebas Unitarias?

Definición

Las pruebas unitarias son un tipo de prueba de software que verifica el funcionamiento de una unidad específica de código, generalmente una función o un método, de manera aislada.

Propósito

Asegurar que cada parte individual del código funcione correctamente por sí sola, antes de integrarla con otras partes del sistema.

Beneficios

- Detectar errores en etapas tempranas del desarrollo.
- Facilitar el mantenimiento del código.
- Ayudar a entender y documentar el comportamiento del código.
- Fomentar un desarrollo más seguro y robusto.

Características y Buenas Prácticas de las Pruebas Unitarias

Automatizables

- No requiere intervención humana

Completas

- Deben cubrir la mayor cantidad de código

Repetibles o Reutilizables

- No deben funcionar para una sola vez

Independientes

- La ejecución de una prueba no debe afectar la ejecución de otra.

Profesionales

- Deben ser consideradas como un entregable más

Otras características

- Fomentan el cambio
  - Permiten hacer las pruebas sobre cambios y asegurarse que los cambios no generan errores.
- Simplifican la integración
  - Permiten llegar a la fase de integración con un grado alto de seguridad de que el código está funcionando correctamente.
- Documenta el código
  - Las propias pruebas son documentación del código
- Separación de la interfaz y la implementación
  - El cambio de la tecnología de interfaz no debe afectar las pruebas.
- Los errores están más acotados y son más fáciles de localizar

Limitaciones

- Las pruebas unitarias NO descubrirán todos los errores de código.
  - No descubrirán errores de integración.
  - Problemas de rendimiento y otros problemas que afectan a todo el sistema en su conjunto.
- Sólo son efectivas si se usan en conjunto con otras pruebas de software.

¿Qué es JUnit 5?

- JUnit 5 es un framework de pruebas unitarias para el lenguaje de programación Java. Es la evolución de JUnit 4 y proporciona un conjunto más flexible y poderoso de herramientas para escribir y ejecutar pruebas.

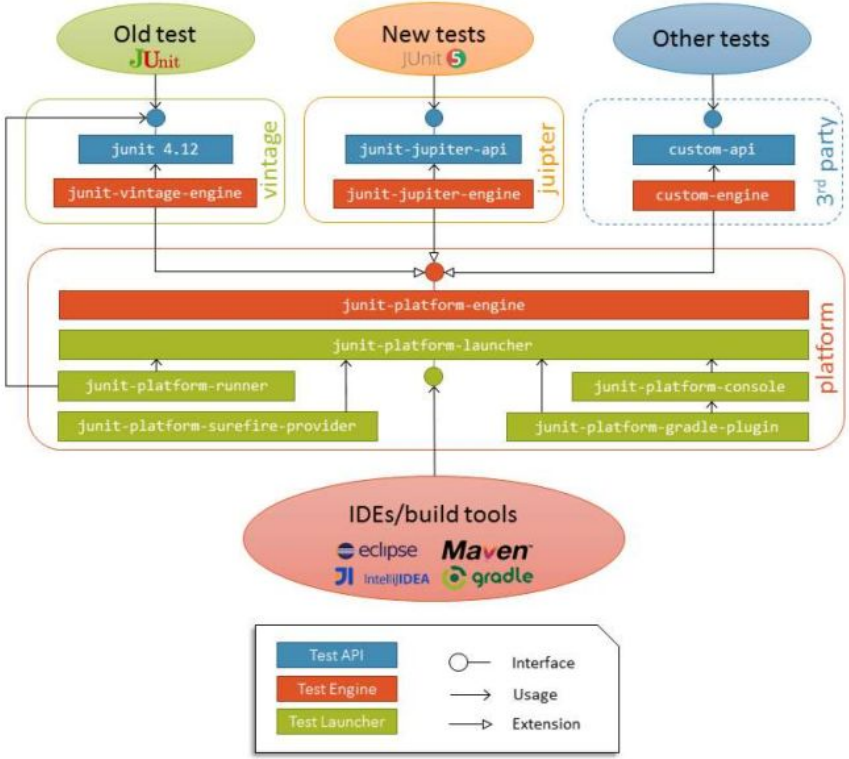
Componentes Principales:

- JUnit Platform: El núcleo que inicia el framework y descubre las pruebas.
- JUnit Jupiter: Nuevo modelo de programación y extensión para escribir pruebas utilizando anotaciones como @Test, @BeforeEach, @AfterEach, etc.
- JUnit Vintage: Soporte para ejecutar pruebas escritas con JUnit 3 y JUnit 4 en JUnit 5.

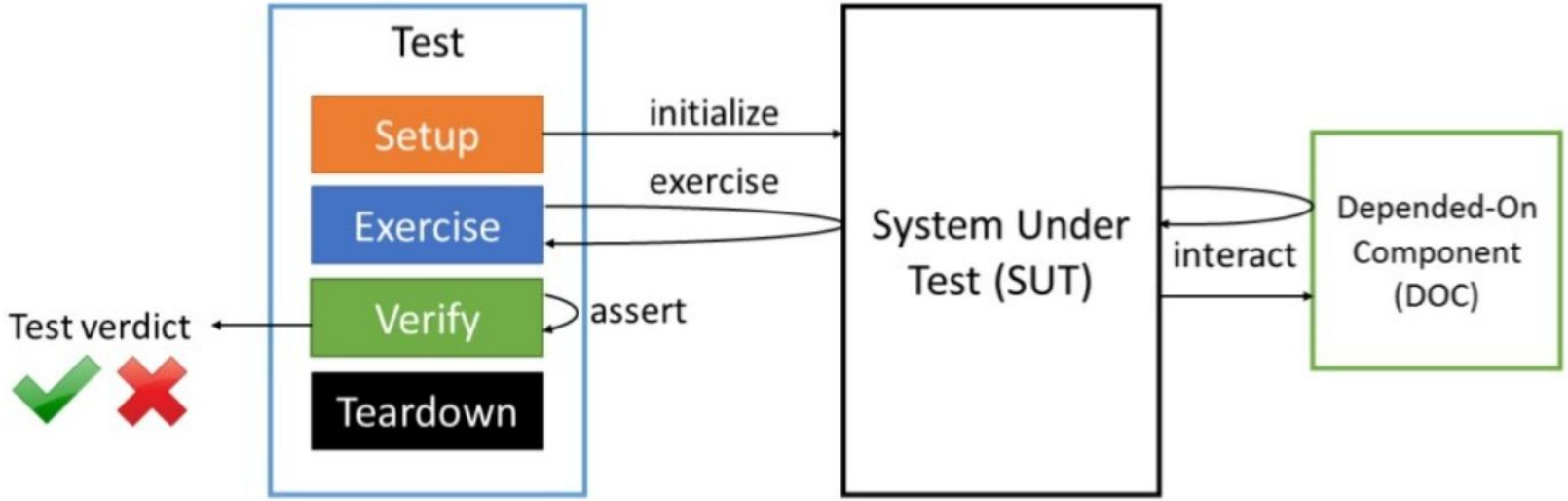
Arquitectura de JUnit 5

- Hay tres tipos de módulos:

1. **Test API**: Módulos usados por *testers* para implementar casos de prueba
2. **Test Engine SPI**: Módulos extendidos para un *framework* de pruebas Java para la ejecución de un modelo concreto de tests
3. **Test Launcher API**: Módulos usados por *clientes programáticos* para el descubrimiento de tests



Esquema de Pruebas Unitarias



Aserciones

- Las aserciones (assertions) en pruebas unitarias son afirmaciones que permiten verificar si un valor o condición cumple con lo esperado en un test.
- Si una aserción falla, la prueba se considera fallida.
- Las aserciones son fundamentales para validar la lógica de una aplicación y asegurar que su comportamiento sea el correcto.

Principales Aserciones en JUnit

- **assertEquals(expected, actual):** Verifica que dos valores sean iguales. Si no lo son, la prueba falla.
- **assertNotEquals(unexpected, actual):** Verifica que dos valores no sean iguales.
- **assertTrue(condition):** Verifica que la condición proporcionada sea verdadera.
- **assertFalse(condition):** Verifica que la condición proporcionada sea falsa.
- **assertNull(object):** Verifica que el objeto proporcionado sea null.
- **assertNotNull(object):** Verifica que el objeto proporcionado no sea null.
- **assertArrayEquals(expectedArray, actualArray):** Verifica que dos arrays sean iguales en tamaño y contenido.
- **assertThrows(expectedType, executable):** Verifica que el código ejecutado lanza una excepción del tipo esperado.
- **assertAll(executables...):** Permite agrupar múltiples aserciones y ejecutarlas todas, mostrando todos los fallos en vez de detenerse en el primero.

Ejemplo

Escribe un método en Java que calcule el precio final de un producto después de aplicar un descuento. El método debe tomar dos argumentos: el precio original y el porcentaje de descuento. Si el porcentaje de descuento es mayor al 100% o menor a 0%, debe lanzar una excepción IllegalArgumentException. Escribe pruebas unitarias usando JUnit para verificar que el método funcione correctamente.

Método que realiza el cálculo

```
public class DiscountCalculator {


    public double calculateFinalPrice(double originalPrice, double discountPercentage) {
        if (discountPercentage < 0 || discountPercentage > 100) {
            throw new IllegalArgumentException("El porcentaje de descuento debe estar entre 0 y 100");
        }
        return originalPrice - (originalPrice * discountPercentage / 100);
    }
}
```

Pruebas Unitarias

```java
class DiscountCalculatorTest {
    DiscountCalculator calculator = new DiscountCalculator();
    @Test
    void testCalculateFinalPrice_ValidDiscount() {
        double result = calculator.calculateFinalPrice(100, 20);
        assertEquals(80, result, 0.01, "El precio final con un 20% de descuento sobre 100 debería ser 80");
    }
    @Test
    void testCalculateFinalPrice_NoDiscount() {
        double result = calculator.calculateFinalPrice(100, 0);
        assertEquals(100, result, 0.01, "El precio final sin descuento debería ser igual al precio original");
    }
    @Test
    void testCalculateFinalPrice_100PercentDiscount() {
        double result = calculator.calculateFinalPrice(100, 100);
        assertEquals(0, result, 0.01, "El precio final con un 100% de descuento debería ser 0");
    }

     @Test
      void testCalculateFinalPrice_InvalidDiscountGreaterThan100() {
          assertThrows(IllegalArgumentException.class, () -> {
              calculator.calculateFinalPrice(100, 150);
          }, "Debería lanzar una IllegalArgumentException si el descuento es mayor a 100%");
      }
      @Test
      void testCalculateFinalPrice_InvalidDiscountLessThan0() {
          assertThrows(IllegalArgumentException.class, () -> {
              calculator.calculateFinalPrice(100, -10);
          }, "Debería lanzar una IllegalArgumentException si el descuento es menor a 0%");
      }
  }
```

Anotaciones destacadas

- @BeforeAll
- @AfterAll
- @BeforeEach
- @AfterEach
- @Test

Nuevo modelo de programación con JUnit 5

- Las clases y métodos de test en JUnit 5 se pueden **etiquetar** usando la anotación @Tag
- Estas etiquetas se pueden usar después para el descubrimiento y ejecución de los test (**filtrado**)

```java
import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.Test;

@Tag("functional")
class FunctionalTest {

    @Test
    void test1() {
        System.out.println("Functional Test 1");
    }

    @Test
    void test2() {
        System.out.println("Functional Test 2");
    }

}
```

```java
import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.Test;

@Tag("non-functional")
class NonFunctionalTest {

    @Test
    @Tag("performance")
    @Tag("load")
    void test1() {
        System.out.println("Non-Functional Test 1 (Performance/Load)");
    }

    @Test
    @Tag("performance")
    @Tag("stress")
    void test2() {
        System.out.println("Non-Functional Test 2 (Performance/Stress)");
    }

    @Test
    @Tag("security")
    void test3() {
        System.out.println("Non-Functional Test 3 (Security)");
    }

    @Test
    @Tag("usability")
    void test4() {
        System.out.println("Non-Functional Test 4 (Usability)");
    }

}
```

Filtrar tags en Maven

```xml
<!-- Maven Surefire plugin to run tests -->
<build>
    <plugins>
        <plugin>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>${maven-surefire-plugin.version}</version>
            <configuration>
                <properties>
                    <includeTags>functional</includeTags>
                    <excludeTags>non-functional</excludeTags>
                </properties>
            </configuration>
            <dependencies>
                <dependency>
                    <groupId>org.junit.platform</groupId>
                    <artifactId>junit-platform-surefire-provider</artifactId>
                    <version>${junit.platform.version}</version>
                </dependency>
                <dependency>
                    <groupId>org.junit.jupiter</groupId>
                    <artifactId>junit-jupiter-engine</artifactId>
                    <version>${junit.jupiter.version}</version>
                </dependency>
            </dependencies>
        </plugin>
    </plugins>
</build>
```

@Disabled

- Permite deshabilitar test a nivel de clase o de método

```java
import org.junit.jupiter.api.Disabled;
import org.junit.jupiter.api.Test;

class DisabledTest {

    @Disabled
    @Test
    void skippedTest() {
    }

}
```

```java
import org.junit.jupiter.api.Disabled;
import org.junit.jupiter.api.Test;

@Disabled("All test in this class will be skipped")
class AllDisabledTest {

    @Test
    void skippedTest1() {
    }

    @Test
    void skippedTest2() {
    }

}
```

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running io.github.bonigarcia.AllDisabledTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 1, Time elapsed:
0.059 sec - in io.github.bonigarcia.AllDisabledTest

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 1
```

@RepeatTest

- Permite repetir un test un número de veces determinado

```java
import org.junit.jupiter.api.RepeatedTest;

class SimpleRepeatedTest {

    @RepeatedTest(5)
    void test() {
        System.out.println("Repeated test");
    }

}
```

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running io.github.bonigarcia.SimpleRepeatedTest
Repeated test
Repeated test
Repeated test
Repeated test
Repeated test
Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
0.11 sec - in io.github.bonigarcia.SimpleRepeatedTest

Results :

Tests run: 5, Failures: 0, Errors: 0, Skipped: 0
```

@ParameterizedTest

- Los pasos para implementar un test parametrizado son:

1. Usar la anotación `@ParameterizedTest` para declarar un test como parametrizado
2. Elegir un proveedor de argumentos (*argument provider*)

Arguments provider	Descripción
`@ValueSource`	Usado para especificar un array de valores `String`, `int`, `long`, o `double`
`@EnumSource`	Usado para especificar valores enumerados (`java.lang.Enum`)
`@MethodSource`	Usado para especificar un método estático de la clase que proporciona un `Stream` de valores
`@CsvSource`	Usado para especificar valores separados por coma, esto es, en formato CSV (*comma-separated values*)
`@CsvFileSource`	Usado para especificar valores en formato CSV en un fichero localizado en el classpath
`@ArgumentsSource`	Usado para especificar una clase que implementa el interfaz `org.junit.jupiter.params.provider.ArgumentsProvider`

Ejemplo con @ValueSource

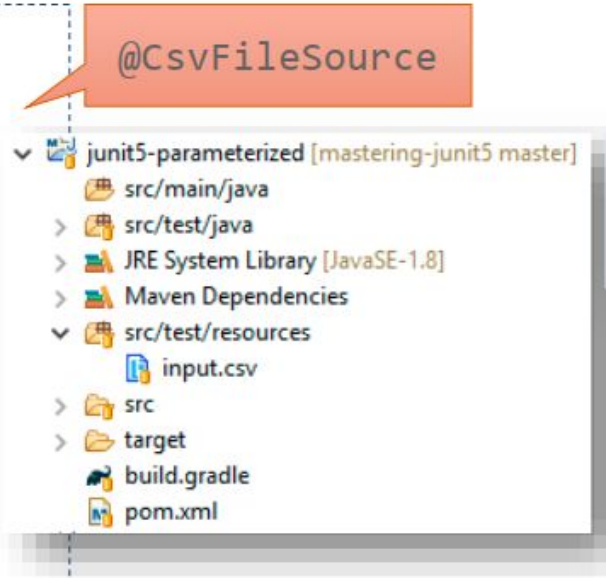
```java
import static org.junit.jupiter.api.Assertions.assertNotNull;

import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.ValueSource;

class ValueSourcePrimitiveTypesParameterizedTest {

    @ParameterizedTest
    @ValueSource(ints = { 0, 1 })
    void testWithInts(int argument) {
        System.out
                .println("Parameterized test with (int) argument: " + argument);
        assertNotNull(argument);
    }

    @ParameterizedTest
    @ValueSource(longs = { 2L, 3L })
    void testWithLongs(long argument) {
        System.out.println(
                "Parameterized test with (long) argument: " + argument);
        assertNotNull(argument);
    }

    @ParameterizedTest
    @ValueSource(doubles = { 4d, 5d })
    void testWithDoubles(double argument) {
        System.out.println(
                "Parameterized test with (double) argument: " + argument);
        assertNotNull(argument);
    }

}
```

@ValueSource

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running io.github.bonigarcia.ValueSourcePrimitiveTypesP
Parameterized test with (int) argument: 0
Parameterized test with (int) argument: 1
Parameterized test with (long) argument: 2
Parameterized test with (long) argument: 3
Parameterized test with (double) argument: 4.0
Parameterized test with (double) argument: 5.0
Tests run: 6, Failures: 0, Errors: 0, Skipped: 0, Time
io.github.bonigarcia.ValueSourcePrimitiveTypesParameter

Results :

Tests run: 6, Failures: 0, Errors: 0, Skipped: 0
```

Ejemplo con @CsvFileSource

```java
import static org.junit.jupiter.api.Assertions.assertNotEquals;
import static org.junit.jupiter.api.Assertions.assertNotNull;

import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvFileSource;

class CsvFileSourceParameterizedTest {

    @ParameterizedTest
    @CsvFileSource(resources = "/input.csv")
    void testWithCsvFileSource(String first, int second) {
        System.out.println("Yet another parameterized test with (String) "
                + first + " and (int) " + second);

        assertNotNull(first);
        assertNotEquals(0, second);
    }

}
```



Referencias

- https://maven.apache.org/archetype/project-info.html
- https://junit.org/junit5/docs/current/user-guide/