



Pruebas Unitarias

UNJu - Programación Orientada a Objetos



Maven y su relación con las pruebas unitarias

¿Qué es Maven?

- Herramienta de automatización de compilación y gestión de proyectos en Java.
- Uso de POM.xml para definir dependencias y configuración.

¿Cómo funciona?

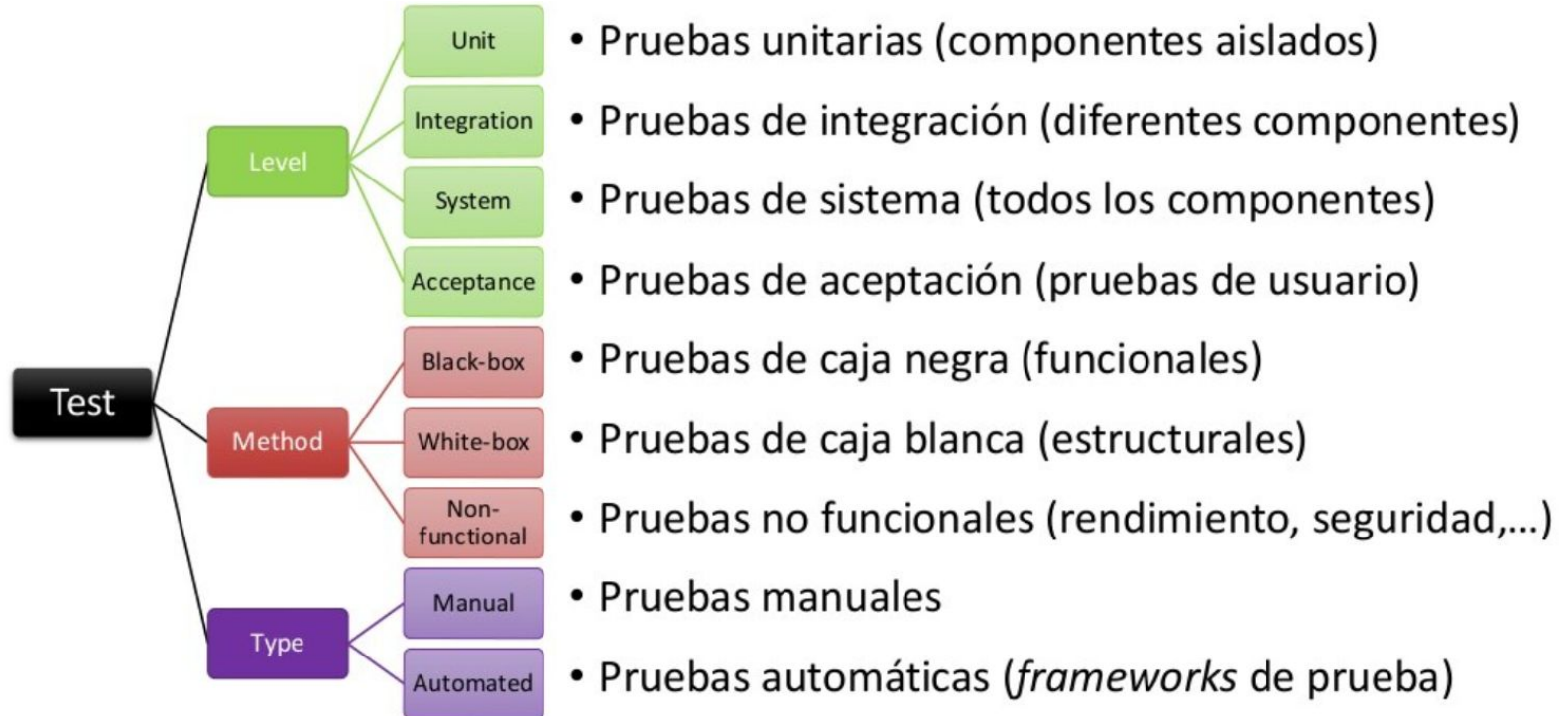
- Descarga automática de dependencias.
- Estándar de estructura de proyectos (src/main/java, src/test/java).

Ciclo de vida

- Fases: validate → compile → test → package → verify → install → deploy.
- El test se integra automáticamente en este ciclo.



Pruebas de software





¿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 integrarse 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

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?

Componentes Principales

- **JUnit Platform:** El núcleo del framework. Proporciona la infraestructura para descubrir y ejecutar pruebas, e integra diferentes motores de ejecución (runners).
- **JUnit Jupiter:** El nuevo modelo de programación de pruebas. Define las anotaciones (@Test, @BeforeEach, @AfterEach, etc.) y la API para escribir y extender pruebas en JUnit 5.
- **JUnit Vintage:** Motor de compatibilidad que permite ejecutar pruebas escritas con JUnit 3 y 4 dentro de JUnit 5..

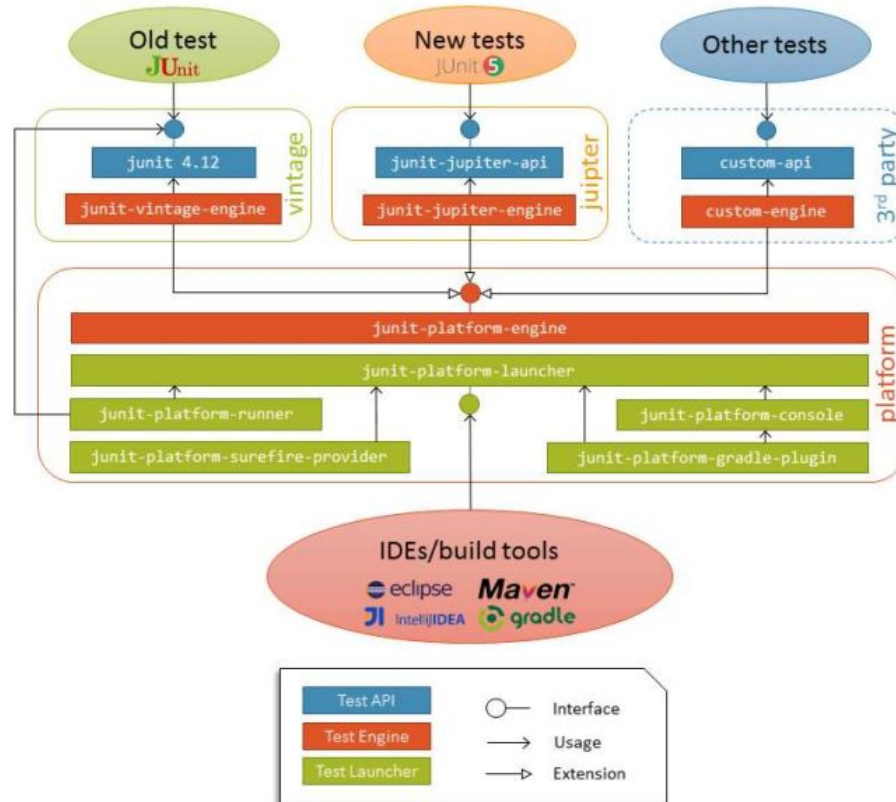


Arquitectura de JUnit 5

- **Test API:**
 - Es la API pública que usan los desarrolladores de pruebas.
- **Test Engine SPI (Service Provider Interface):**
 - Es una interfaz para proveedores de motores de pruebas.
 - Permite que distintos motores (como JUnit Jupiter, JUnit Vintage, o incluso de terceros como Cucumber) se integren con la plataforma.
- **Test Launcher API**
 - Es la API que usan las herramientas externas (IDE, build tools como Maven/Gradle, o CI/CD) para lanzar pruebas en la plataforma.
 - Se encarga de descubrir, filtrar y ejecutar pruebas, además de publicar resultados y notificaciones.

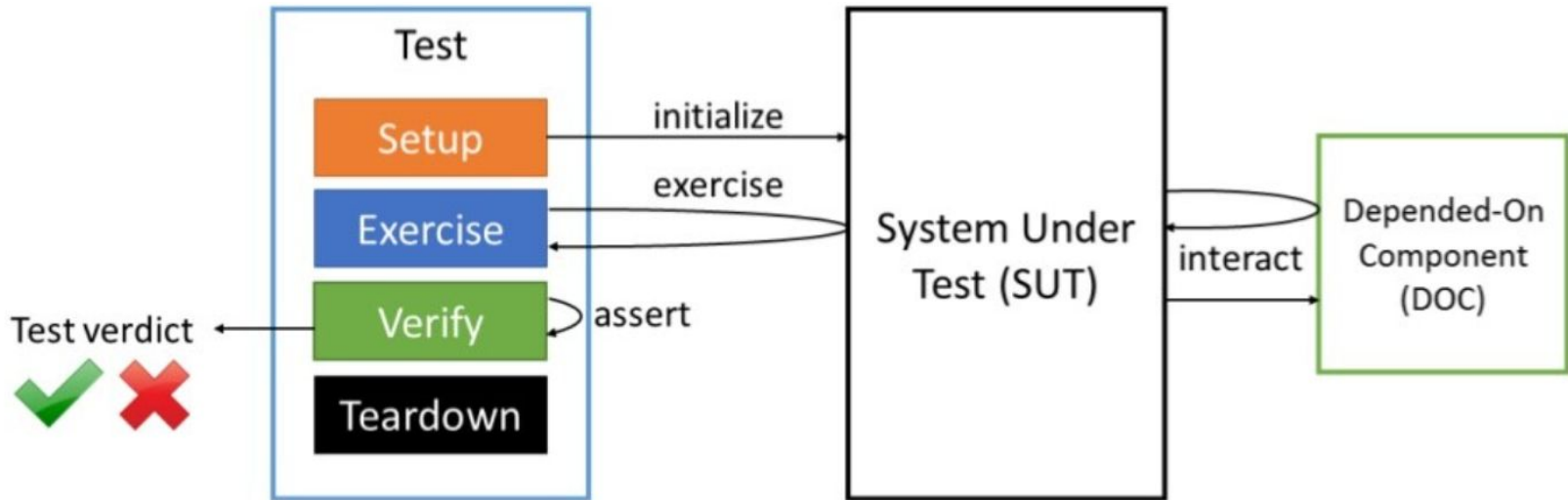


Arquitectura de JUnit 5





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

```
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

- Ciclo de vida:
 - `@BeforeAll`, `@AfterAll`
 - `@BeforeEach`, `@AfterEach`
- Ejecución de pruebas:
 - `@Test`
 - `@RepeatedTest(n)`
- Organización y documentación:
 - `@DisplayName("Texto descriptivo")`
 - `@Tag("etiqueta")`
 - `@Disabled`
- Tests parametrizados:
 - `@ValueSource` (valores simples).
 - `@CsvSource` / `@CsvFileSource` (datos tabulares).



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**)

```
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");
    }

}
```

```
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

```
<!-- 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>
```

ork me on GitHub



@Disabled

- Permite deshabilitar test a nivel de clase o de método

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

class DisabledTest {

    @Disabled
    @Test
    void skippedTest() {
    }

}
```

```
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

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



Ejemplo con @ValueSource

```
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

```
TESTS
-----
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

```
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);
    }
}
```

@CsvFileSource

- ▼ junit5-parameterized [mastering-junit5 master]
 - src/main/java
 - > src/test/java
 - > JRE System Library [JavaSE-1.8]
 - > Maven Dependencies
 - ▼ src/test/resources
 - input.csv
 - > src
 - > target
 - build.gradle
 - pom.xml



Referencias

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