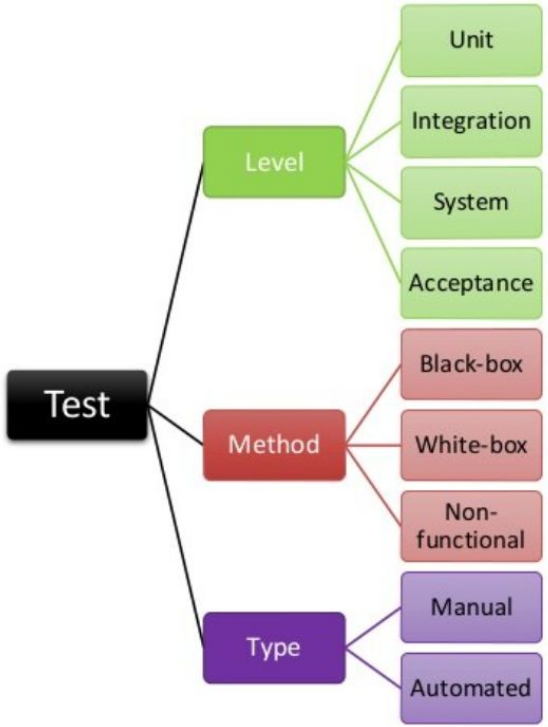
Pruebas Unitarias

UNJu - Desarrollo y Arquitecturas Avanzadas

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

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?

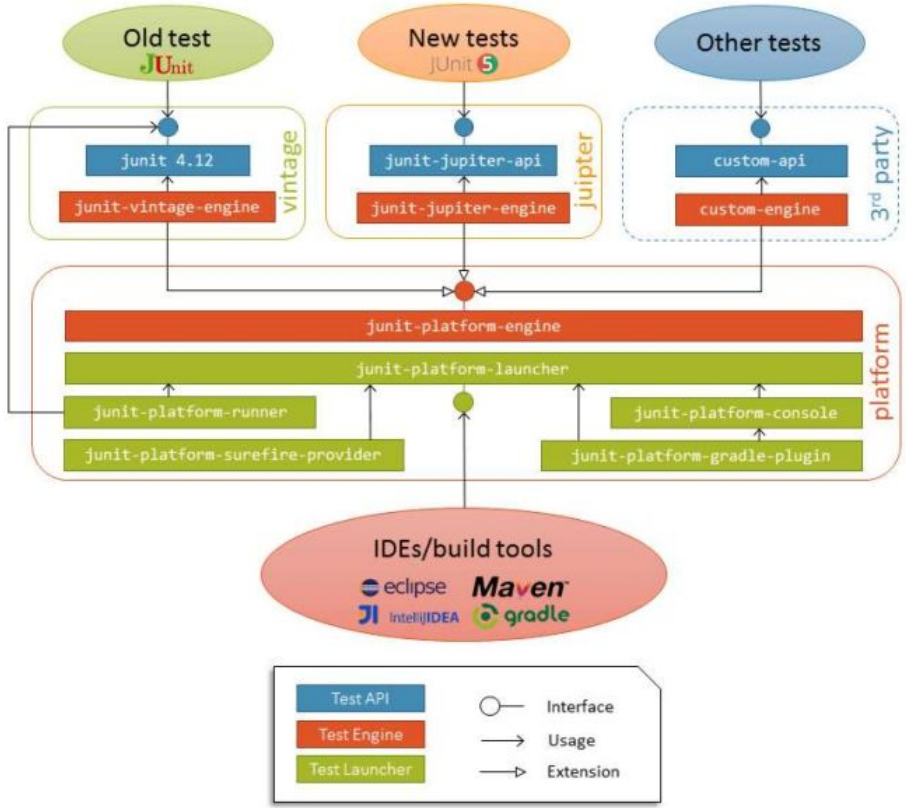
Componentes Principales

- **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..

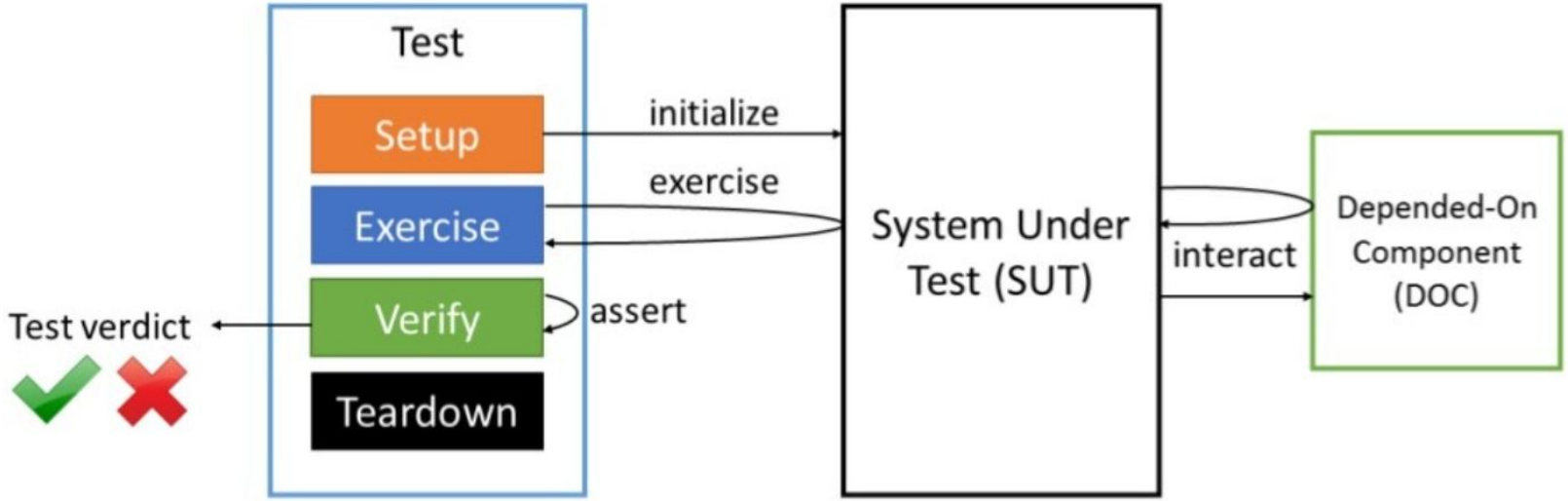
Arquitectura de JUnit 5

- **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

```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

Referencias

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