Pruebas Unitarias

Entornos de desarrollo

Diapositivas realizadas por Aitor Ventura Edo

IES El Caminás
Curso 2025-2026

INTRODUCCIÓN

Pruebas unitarias

Las pruebas unitarias verifican el correcto funcionamiento de módulos individuales del código, asegurando que cada parte del programa funcione de manera independiente antes de la integración del sistema completo.

Estas pruebas deben cumplir con ciertos requisitos:

- Automatizables: No requieren intervención manual.
- Completas: Cubren la mayor cantidad posible de código.
- Reutilizables: No están limitadas a una única ejecución.
- Independientes: Una prueba no debe influir en otra.
- Profesionales: Documentadas y desarrolladas con la misma calidad que el código.

Objetivos y ventajas

- **Fomentan el cambio:** Facilitan la modificación del código con confianza al garantizar que los cambios no introducen errores.
- **Simplifican la integración:** Aumentan la seguridad antes de combinar componentes.
- **Documentan el código:** Las pruebas muestran cómo usar las unidades probadas.
- **Separan interfaz e implementación:** Permiten modificar la implementación o la interfaz sin afectar al otro.
- **Facilitan la localización de errores:** Ayudan a identificar problemas de manera aislada.

JUNIT

¿Qué es JUnit y para qué se usa?

- JUnit es un framework de pruebas unitarias para Java.
- Permite verificar que cada unidad de código (método o clase) funciona como se espera.
- Ayuda a detectar errores en etapas tempranas del desarrollo.
- Es parte del ciclo de desarrollo ágil y TDD (Desarrollo guiado por pruebas).
- Se usa con herramientas como Maven o Gradle para integración continua.

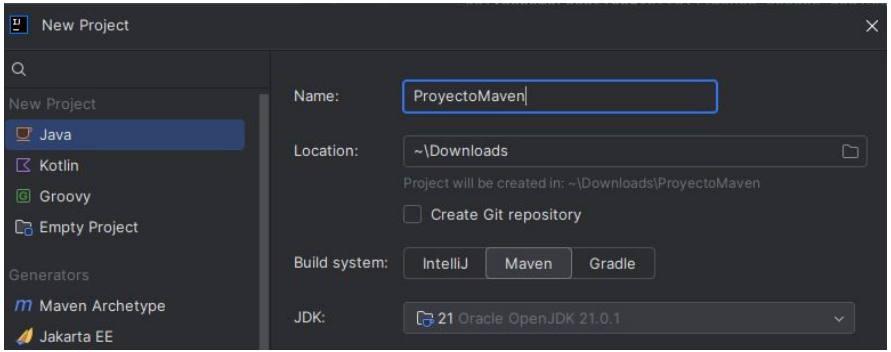
Conceptos clave de JUnit

- Prueba unitaria: Verifica una funcionalidad específica en aislamiento.
- Asserts: Validan el comportamiento esperado frente al obtenido.
- Anotaciones: Definen el ciclo de vida de las pruebas.
- Clase de prueba: Contiene métodos que verifican diferentes aspectos del código.

Configuración básica de JUnit

Utilizaremos JUnit en nuestros proyectos Maven. Un proyecto Maven es un enfoque estandarizado para la gestión y construcción de proyectos de software en Java.

Maven es una herramienta de automatización de proyectos que utiliza un archivo llamado pom.xml (Project Object Model) como núcleo para describir la estructura, dependencias y configuraciones del proyecto.



Configuración básica de JUnit

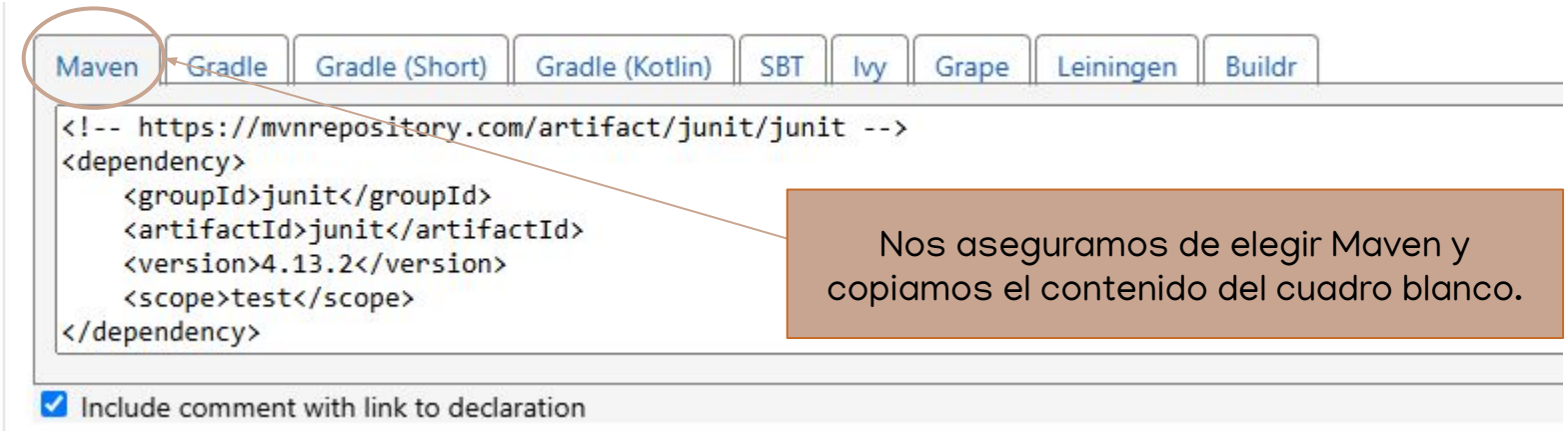
Configuración básica de JUnit

Para poder utilizar JUnit tenemos que importar la dependencia. Para ello buscamos la dependencia en el buscador.

Seleccionamos la web oficial de repositorios de maven, también la puedes encontrar aquí.

Elegimos la versión más reciente

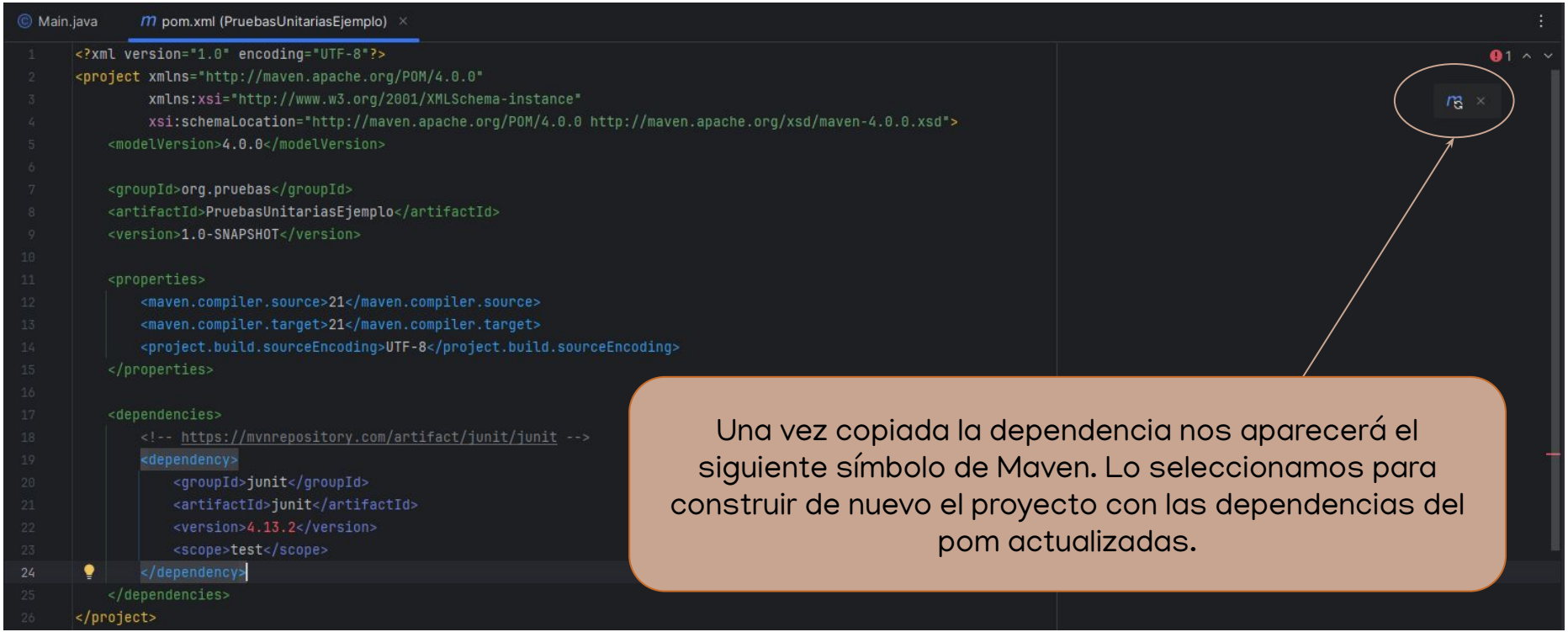
Configuración básica de JUnit



Pegamos la dependencia en nuestro pom, dentro de las etiquetas <dependencies> <dependencies>.

Aquí se meterán todas las dependencias que necesitemos en nuestro proyecto.

Configuración básica de JUnit



Anotaciones esenciales en JUnit

- **@Test**: Marca un método como prueba.
- **@BeforeEach**: Se ejecuta antes de cada prueba.
- **@AfterEach**: Se ejecuta después de cada prueba.
- **@BeforeAll**: Se ejecuta una vez al inicio de la suite.
- **@AfterAll**: Se ejecuta una vez al final de la suite.

Principales métodos

- assertEquals(expected, actual): Verifica igualdad.
- assertTrue(condition): Verifica que una condición sea verdadera.
- assertFalse(condition): Verifica que una condición sea falsa.
- assertNull(object): Verifica que un objeto sea null.
- assertNotNull(object): Verifica que un objeto no sea null.
- assertThrows(TipoDeExcepcion.class, () -> { // Código que debería lanzar la excepción }); : Verifica que se lanza TipoDeExcepcion.

Ejemplo

Ejemplo de gestor de tareas

Tests parametrizados en JUnit

Tests parametrizados

Los tests parametrizados en JUnit permiten ejecutar un método de prueba varias veces con diferentes conjuntos de datos de entrada.

Esto elimina la necesidad de escribir pruebas individuales para cada combinación de parámetros, lo que hace que los tests sean más compactos, reutilizables y fáciles de mantener.

Ventajas de los tests parametrizados

Evitación de redundancia: En lugar de escribir múltiples métodos de prueba con pequeños cambios en los datos, puedes definirlos una sola vez y probar varias combinaciones de entradas.

Cobertura completa: Es ideal para probar casos límite, valores extremos y diversas combinaciones de entrada.

Mantenibilidad: Facilita la gestión de pruebas porque solo necesitas actualizar un único método si cambian los requisitos.

Anotaciones clave

1. @ParameterizedTest:
   Indica que un método es un test parametrizado.
2. **Fuentes de datos para parametrización:**
   - @ValueSource: Proporciona un único conjunto de valores de entrada de un tipo.
   - @CsvSource: Proporciona múltiples columnas de datos, separadas por comas.
   - @CsvFileSource: Permite leer datos desde un archivo CSV.
   - @MethodSource: Utiliza un método para proporcionar datos dinámicos o más complejos.

Ejemplo Calculadora

Dado el siguiente ejemplo:

```
public class Calculadora {
    public int sumar(int a, int b) {
        if (a < 0 || b < 0) {
            throw new IllegalArgumentException("No números negativos");
        }
        return a + b;
    }
}
```

Ejemplo Calculadora

Si probamos el método tal sin usar la parametrización:

```
@Test
void testSumarCasoValido() {
    assertEquals(5, calculadora.sumar(2, 3));
    assertEquals(7, calculadora.sumar(3, 4));
}

@Test
void testSumarConNumeroNegativo() {
    assertThrows(IllegalArgumentException.class, () -> calculadora.sumar(-1, 3));
    assertThrows(IllegalArgumentException.class, () -> calculadora.sumar(3, -4));
}
```

Ejemplo Calculadora

Usando parametrización:

```
@ParameterizedTest
@CsvSource({
    "2, 3, 5",
    "3, 4, 7"
})
void testSumarCasosValidos(int a, int b, int esperado) {
     assertEquals(esperado, calculadora.sumar(a, b));
}

@ParameterizedTest
@CsvSource({
    "-1, 3",
    "3, -4"
})
void testSumarConNumerosNegativos(int a, int b) {
    assertThrows(IllegalArgumentException.class, () -> calculadora.sumar(a, b));
}
```

Ejemplo Número par

Dado el siguiente ejemplo:

```
public class Validador {
    public boolean esPar(int numero) {
        if (numero < 0) {
            throw new IllegalArgumentException("El número no puede ser negativo");
        }
        return numero % 2 == 0;
    }
}
```

Ejemplo Número par

Si probamos el método tal sin usar la parametrización:

```
@Test
void testEsParConNumeroValido() {
   assertTrue(validador.esPar(2));
  assertTrue(validador.esPar(4));
}

@Test
void testEsParConNumeroNegativo() {
   assertThrows(IllegalArgumentException.class, () -> validador.esPar(-2));
  assertThrows(IllegalArgumentException.class, () -> validador.esPar(-4));
}
```

Ejemplo Número par

Usando parametrización:

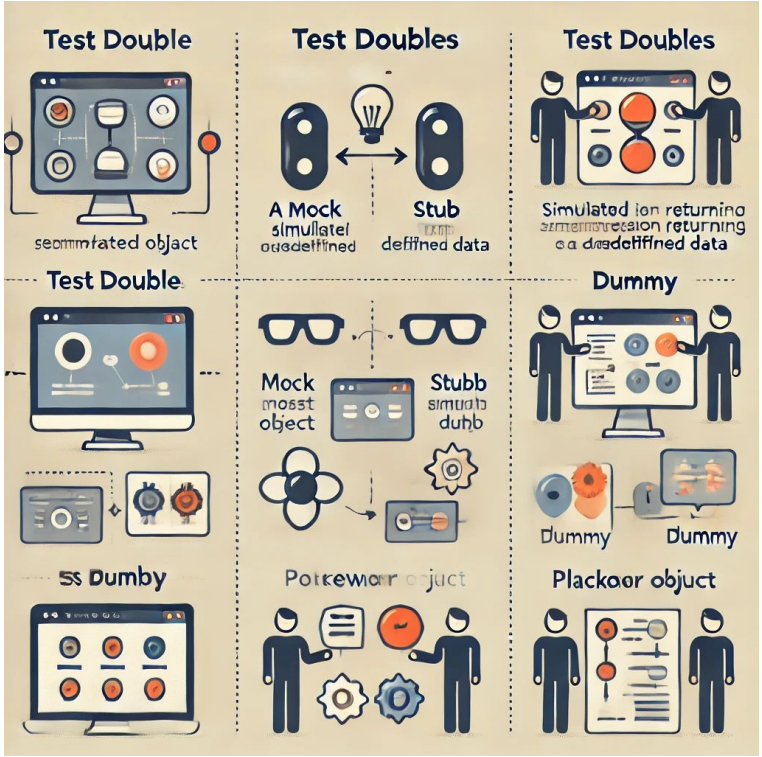
```
@ParameterizedTest
@ValueSource(ints = {2, 4, 6, 8})
void testEsParConNumerosValidos(int numero) {
   assertTrue(validador.esPar(numero));
}

@ParameterizedTest
@ValueSource(ints = {-2, -4, -6})
void testEsParConNumerosNegativos(int numero) {
    assertThrows(IllegalArgumentException.class, () -> validador.esPar(numero));
}
```

Dobles de prueba

Dobles de prueba

Los dobles de prueba son objetos que reemplazan partes del sistema que normalmente no queremos incluir en nuestras pruebas unitarias.

Esto puede incluir bases de datos, servicios web, archivos o cualquier otra dependencia externa.



Tipos de dobles de prueba comunes

Mocks (Simulaciones):

- Son objetos "falsos" que simulan el comportamiento de dependencias reales.
- Te permiten definir qué deberían devolver ciertos métodos.
- Puedes verificar si los métodos fueron llamados y con qué parámetros.

Stubs (Códigos simulados):

- Son versiones simplificadas de dependencias reales que devuelven datos predefinidos.

Dummies (Valores ficticios):

- Son objetos que no tienen funcionalidad pero se necesitan para ejecutar el código.

Uso de Mocks en IntelliJ con Mockito

Asegúrate de tener el paquete de Mockito (añade la dependencia al pom.xml)

https://mvnrepository.com/artifact/org.mockito/mockito-core/5.15.2

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>5.15.2</version>
    <scope>test</scope>
</dependency>
```

Ejemplo

Tenemos una clase Calculadora con dos operaciones, sumar y multiplicar:

```
public class Calculadora {
    public int sumar(int a, int b) {
        return a + b;
    }

    public int multiplicar(int a, int b) {
        return a * b;
    }
}
```

Ejemplo

Y tenemos un servicio que hace el uso de operaciones de la clase calculadora:

```
public class CalculadoraService {
    private Calculadora calculadora;

    public CalculadoraService(Calculadora calculadora) {
        this.calculadora = calculadora;
    }

    public int dobleDeLaSuma(int a, int b) {
        int suma = calculadora.sumar(a, b);
        return calculadora.multiplicar(suma, 2);
    }
}
```

¿Por qué necesitamos un mock de Calculadora?

Separar la lógica a probar de las dependencias externas:

- En este caso, queremos probar el método dobleDeLaSuma de la clase CalculadoraService.
- Este método depende de la clase Calculadora para realizar dos operaciones (sumar y multiplicar).
- Si no usamos un mock, estaríamos probando tanto la lógica de CalculadoraService como la implementación real de Calculadora.

¿Por qué necesitamos un mock de Calculadora?

Problema:

Si hay un error en la clase Calculadora, el test podría fallar, aunque CalculadoraService esté funcionando correctamente. Esto haría más difícil identificar la fuente del problema.

Controlar los resultados de las dependencias:

- Al usar un mock, podemos controlar exactamente qué resultados devuelven los métodos de la clase Calculadora (por ejemplo, sumar y multiplicar).
- Esto nos permite asegurarnos de que estamos probando solo la lógica de CalculadoraService, sin importar cómo funcione la clase Calculadora.

Un cocinero y sus herramientas

Imaginemos que estamos probando a un cocinero (CalculadoraService) que usa herramientas (Calculadora) para hacer su trabajo:

- Si queremos evaluar las habilidades del cocinero, no probamos la calidad de sus herramientas. En lugar de eso, le damos herramientas simuladas que sabemos que funcionan perfectamente.
- Así, podemos concentrarnos en evaluar lo que realmente hace el cocinero, sin preocuparnos por posibles defectos en las herramientas.

```
public class CalculadoraServiceTest

    public void testDobleDeLaSuma()
        // 1. Crear el mock de la clase Calcu
        Calculadora mockCalculadora = mock(Calculadora.c

        // 2. Configurar el mock para devolver valores contr
        when(mockCalculadora.sumar(3, 2)).thenRetu
        when(mockCalculadora.multiplicar(5, 2)).thenRetur

        // 3. Crear una instancia de CalculadoraService con el
        CalculadoraService service = new CalculadoraService(mockCalcula

        // 4. Llamar al método a
        int resultado = service.dobleDeLaSuma(3,

        // 5. Verificar que el resultado es el es
        assertEquals(10, resul

}
```