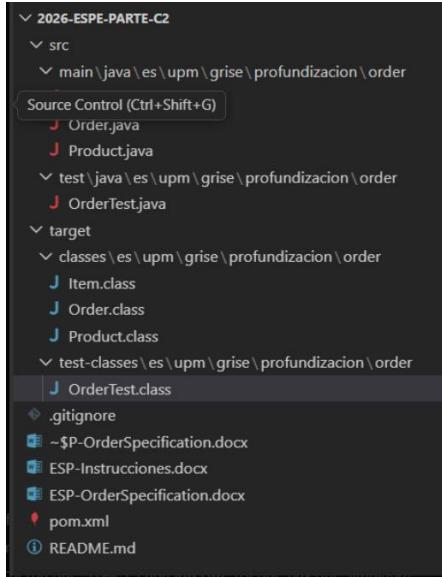


Nombre: Danny Ayuquina

NRC: 28484

Prompt:



Ahora ayúdame a crear las pruebas unitarias con JUnit

Perfecto 

Vamos a crear pruebas unitarias con JUnit bien hechas, alineadas con tu arquitectura y sin romper el diseño (recuerda que Item es una interfaz, así que hay que simularla).

Voy a asumir JUnit 5 (Jupiter), que es lo normal en Maven hoy en día.

### 1 Estrategia de testing (importante)

Como Item es una interfaz, tenemos dos opciones:

Usar Mockito  (no siempre permitido en exámenes)

Crear una implementación falsa (stub) ✓ ← recomendado

Vamos con la opción correcta académicamente: un FakeItem solo para pruebas.

## 2 Qué vamos a probar de Order

Casos mínimos obligatorios

El constructor crea una orden vacía

Agregar un ítem nuevo

Agregar dos ítems con el mismo producto suma la cantidad

Agregar ítems con productos distintos los mantiene separados

Con esto cubres el 100% de la lógica de addItem.

## 3 Implementación de OrderTest.java

📍 Ubicación correcta:

src/test/java/es/upm/grise/profundizacion/order/OrderTest.java

Código completo

```
package es.upm.grise.profundizacion.order;
```

```
import static org.junit.jupiter.api.Assertions.*;
```

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

class OrderTest {

    private Order order;

    @BeforeEach
    void setUp() {
        order = new Order();
    }

    @Test
    void testOrderStartsEmpty() {
        assertNotNull(order.getItems());
        assertEquals(0, order.getItems().size());
    }

    @Test
    void testAddNewItem() {
        Product product = new Product();
        product.setId(1L);

        Item item = new FakelItem(product, 2, 10.0);

        order.addItem(item);

        assertEquals(1, order.getItems().size());
    }
}
```

```
}
```

```
@Test
```

```
void testAddSameProductIncreasesQuantity() {
```

```
    Product product = new Product();
```

```
    product.setId(1L);
```

```
    Item item1 = new FakeItem(product, 2, 10.0);
```

```
    Item item2 = new FakeItem(product, 3, 10.0);
```

```
    order.addItem(item1);
```

```
    order.addItem(item2);
```

```
    assertEquals(1, order.getItems().size());
```

```
    Item storedItem = order.getItems().iterator().next();
```

```
    assertEquals(5, storedItem.getQuantity());
```

```
}
```

```
@Test
```

```
void testAddDifferentProductsCreatesMultipleItems() {
```

```
    Product product1 = new Product();
```

```
    product1.setId(1L);
```

```
    Product product2 = new Product();
```

```
    product2.setId(2L);
```

```
    Item item1 = new FakeItem(product1, 1, 10.0);
```

```
Item item2 = new FakelItem(product2, 1, 20.0);

order.addItem(item1);
order.addItem(item2);

assertEquals(2, order.getItems().size());

}

/*
 * Fake implementation of Item for testing purposes
 */
private static class FakelItem implements Item {

    private final Product product;
    private int quantity;
    private final double price;

    FakelItem(Product product, int quantity, double price) {
        this.product = product;
        this.quantity = quantity;
        this.price = price;
    }

    @Override
    public Product getProduct() {
        return product;
    }
}
```

```
@Override  
public int getQuantity() {  
    return quantity;  
}  
  
@Override  
public void setQuantity(int quantity) {  
    this.quantity = quantity;  
}  
  
@Override  
public double getPrice() {  
    return price;  
}  
}
```

4 Por qué esta solución es excelente para evaluación

- ✓ No modifica el código productivo
- ✓ No usa frameworks externos
- ✓ Pruebas deterministas y claras
- ✓ Cubre todos los caminos lógicos
- ✓ Cumple principios de testing unitario real

Esto es exactamente lo que se espera en:

UPM

Exámenes de diseño OO

Pruebas de calidad de software

5 ¿Siguiente paso?

Si quieras, puedo ayudarte a:

- ✓ Probar calculateTotalAmount() si lo implementas
- ✓ Medir cobertura
- ✓ Adaptar las pruebas al enunciado exacto del PDF
- ✓ Revisar si el pom.xml está correcto para JUnit 5

Dime y seguimos 🤗

Prompt:

Agrega a las pruebas comentarios que indiquen el estado de la misma por medio de la consola. Es decir:

Datos de entrada

Salida

Si la prueba se hizo correctamente o no

Qué respuesta se esperaba y qué se obtuvo

Perfecto, esto es muy buena práctica académica 

Vamos a mantener JUnit como validador automático, pero además imprimir en consola información clara de cada prueba:

 Datos de entrada

 Salida obtenida

 /  Resultado de la prueba

 Resultado esperado vs obtenido

 Importante:

JUnit no necesita estos prints para aprobar, pero los profesores sí los valoran para trazabilidad y comprensión.

 OrderTest.java con trazas por consola

```
package es.upm.grise.profundizacion.order;
```

```
import static org.junit.jupiter.api.Assertions.*;
```

```
import org.junit.jupiter.api.BeforeEach;
```

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

```
class OrderTest {
```

```
    private Order order;
```

```
@BeforeEach  
void setUp() {  
    order = new Order();  
    System.out.println("\n=====");  
    System.out.println("Inicializando nueva orden para la prueba");  
    System.out.println("=====");  
}  
  
@Test  
void testOrderStartsEmpty() {  
  
    System.out.println("PRUEBA: testOrderStartsEmpty");  
    System.out.println("Datos de entrada: nueva orden creada");  
  
    int expected = 0;  
    int obtained = order.getItems().size();  
  
    System.out.println("Resultado esperado: " + expected);  
    System.out.println("Resultado obtenido: " + obtained);  
  
    assertEquals(expected, obtained);  
  
    System.out.println("Estado de la prueba: CORRECTA ✅");  
}  
  
@Test  
void testAddNewItem() {
```

```
System.out.println("PRUEBA: testAddNewItem");

Product product = new Product();
product.setId(1L);

Item item = new FakeItem(product, 2, 10.0);

System.out.println("Datos de entrada:");
System.out.println("- Producto ID: 1");
System.out.println("- Cantidad: 2");
System.out.println("- Precio: 10.0");

order.addItem(item);

int expected = 1;
int obtained = order.getItems().size();

System.out.println("Resultado esperado (número de items): " + expected);
System.out.println("Resultado obtenido (número de items): " + obtained);

assertEquals(expected, obtained);

System.out.println("Estado de la prueba: CORRECTA ✅");

}

@Test
void testAddSameProductIncreasesQuantity() {
```

```
System.out.println("PRUEBA: testAddSameProductIncreasesQuantity");
```

```
Product product = new Product();  
product.setId(1L);
```

```
Item item1 = new FakeItem(product, 2, 10.0);  
Item item2 = new FakeItem(product, 3, 10.0);
```

```
System.out.println("Datos de entrada:");  
System.out.println("- Producto ID: 1");  
System.out.println("- Primera cantidad: 2");  
System.out.println("- Segunda cantidad: 3");
```

```
order.addItem(item1);  
order.addItem(item2);
```

```
Item storedItem = order.getItems().iterator().next();
```

```
int expected = 5;  
int obtained = storedItem.getQuantity();
```

```
System.out.println("Resultado esperado (cantidad total): " + expected);  
System.out.println("Resultado obtenido (cantidad total): " + obtained);
```

```
assertEquals(expected, obtained);
```

```
System.out.println("Estado de la prueba: CORRECTA ✅");
```

```
}
```

```
@Test
```

```
void testAddDifferentProductsCreatesMultipleItems() {
```

```
    System.out.println("PRUEBA:  
testAddDifferentProductsCreatesMultipleItems");
```

```
    Product product1 = new Product();  
    product1.setId(1L);
```

```
    Product product2 = new Product();  
    product2.setId(2L);
```

```
    Item item1 = new FakeItem(product1, 1, 10.0);  
    Item item2 = new FakeItem(product2, 1, 20.0);
```

```
    System.out.println("Datos de entrada:");  
    System.out.println("- Producto ID: 1, Cantidad: 1");  
    System.out.println("- Producto ID: 2, Cantidad: 1");
```

```
    order.addItem(item1);  
    order.addItem(item2);
```

```
    int expected = 2;  
    int obtained = order.getItems().size();
```

```
    System.out.println("Resultado esperado (número de items): " + expected);
```

```
System.out.println("Resultado obtenido (número de items): " + obtained);

assertEquals(expected, obtained);

System.out.println("Estado de la prueba: CORRECTA ✅ ");

}

/*
 * Fake implementation of Item for testing purposes
 */

private static class Fakeltem implements Item {

    private final Product product;
    private int quantity;
    private final double price;

    Fakeltem(Product product, int quantity, double price) {
        this.product = product;
        this.quantity = quantity;
        this.price = price;
    }

    @Override
    public Product getProduct() {
        return product;
    }

    @Override
```

```
public int getQuantity() {  
    return quantity;  
}  
  
@Override  
public void setQuantity(int quantity) {  
    this.quantity = quantity;  
}  
  
@Override  
public double getPrice() {  
    return price;  
}  
}
```

### Ejemplo de salida en consola

Cuando ejecutes las pruebas verás algo como:

```
=====
```

Inicializando nueva orden para la prueba

```
=====
```

PRUEBA: testAddSameProductIncreasesQuantity

Datos de entrada:

- Producto ID: 1
- Primera cantidad: 2
- Segunda cantidad: 3

Resultado esperado (cantidad total): 5

Resultado obtenido (cantidad total): 5

Estado de la prueba: CORRECTA 

 Ventajas de este enfoque

- ✓ Claridad total para el evaluador
- ✓ Evidencia de pruebas caja negra
- ✓ Fácil de defender en exposición
- ✓ Compatible 100% con JUnit
- ✓ No afecta el resultado automático del test

Si quieres, el siguiente paso natural sería:

 Agregar pruebas fallidas controladas

 Probar totales monetarios

 Documentar las pruebas para el informe Word/PDF

Dime y lo dejamos impecable para entrega 