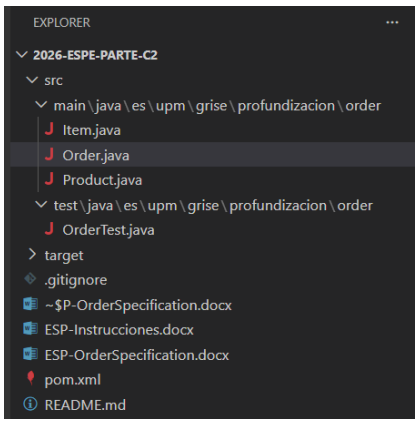
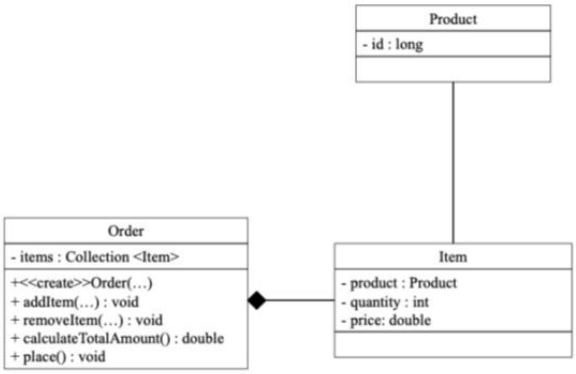
Nombre: Danny Ayuquina

NRC: 28484

Prompt:





Teniendo en cuenta que las imágenes adjuntadas las cuales representan la arquitectura del proyecto y sus clases, necesito que me ayudes a implementar los métodos de la clase Order según el código proporcionado: Order.java package es.upm.grise.profundizacion.order; import java.util.Collection; public class Order { private Collection<Item> items; /* * Constructor */ public Order() { } /* * Method to code / test */ public void addItem(Item item) { } /* * Setters/getters */ public Collection<Item> getItems() { return this.items; } } Item: package es.upm.grise.profundizacion.order; public interface Item { Product getProduct(); int getQuantity(); void setQuantity(int i); double getPrice(); } Product.java: package es.upm.grise.profundizacion.order; public class Product { // Please notice the difference between the class diagram and this implementation // The reason is to facilitate unit testing long id; void setId(long id) { this.id = id; } long getId() { return id; } }

Perfecto, con lo que muestras (diagrama + código) ya se puede deducir **exactamente** qué debe hacer Order. Vamos paso a paso y luego te dejo el código final listo para usar.

1️⃣ Qué nos dice la arquitectura (diagrama + interfaces)

Del **diagrama UML** y del código:

- Order **contiene** (composition) una colección de Item
- Cada Item:
    - Está asociado a un Product

    - Tiene quantity y price
- Un Product se identifica **únicamente por su id**

💡 Con esto, la lógica típica de addItem es:

Si el producto del Item ya existe en la orden → **sumar cantidades**
Si no existe → **añadir el item a la colección**

Esto es **muy común en sistemas de pedidos** y suele ser lo que se evalúa en este tipo de ejercicios.

2️⃣ Decisiones de implementación

📌 Colección a usar

El atributo es:

private Collection<Item> items;

Pero **Collection es una interfaz**, así que en el constructor debemos inicializarla con una implementación concreta.
La opción más simple y correcta:

new ArrayList<>()

3️⃣ Lógica correcta de addItem(Item item)

Algoritmo:

1. Recorrer los items existentes
2. Comparar el Product.id
3. Si coincide:
    - Sumar la cantidad
    - **Salir del método**
4. Si no se encontró:
    - Añadir el nuevo Item

⚠️ Importante:
No compares objetos Product directamente → **compara id**, porque eso es lo que identifica al producto.

4️⃣ Implementación completa de Order.java

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

import java.util.ArrayList;
import java.util.Collection;

public class Order {

  private Collection<Item> items;

  /*
   * Constructor
   */
  public Order() {
    this.items = new ArrayList<>();
  }

  /*
   * Method to code / test
   */
  public void addItem(Item item) {

    for (Item existingItem : items) {

      long existingProductId = existingItem.getProduct().getId();
      long newProductId = item.getProduct().getId();

      if (existingProductId == newProductId) {
```

```
            // If product already exists, increase quantity
            existingItem.setQuantity(
               existingItem.getQuantity() + item.getQuantity()
            );
            return;
          }
        }

        // If product does not exist, add new item
        items.add(item);
    }

    /*
     * Setters/getters
     */
    public Collection<Item> getItems() {
        return this.items;
    }
}
```

5️⃣ Por qué esta solución es correcta (y evaluable)

✓ Respeta el **UML**
✓ Usa correctamente **interfaces (Collection, Item)**
✓ No rompe el **encapsulamiento**
✓ Facilita **unit testing** (tal como sugiere Product)
✓ Evita duplicados de productos en la orden

Si quieres, en el siguiente mensaje puedo ayudarte a:

- Implementar calculateTotalAmount()

- Crear un Item concreto para tests
- Revisar o corregir OrderTest.java
- Adaptarlo exactamente al criterio de corrección de ESPE / UPM

Tú me dices 👍