Composite

Composite es un patrón de diseño estructural que te permite componer objetos en estructuras de árbol y trabajar con esas estructuras como si fueran objetos individuales.
El patrón de diseño Composite (Composite Pattern) es uno de los 23 patrones de diseño “GoF” para el desarrollo de software que fueron publicados en 1994 por Erich Gamma, Richard Helm, Ralph Johnson, y John Vlissides, los llamados “Gang of Four” o la banda de los cuatro. Así como el patrón Facade y el patrón Decorator, se trata de un patrón de diseño que agrupa objetos complejos y clases en estructuras mayores.

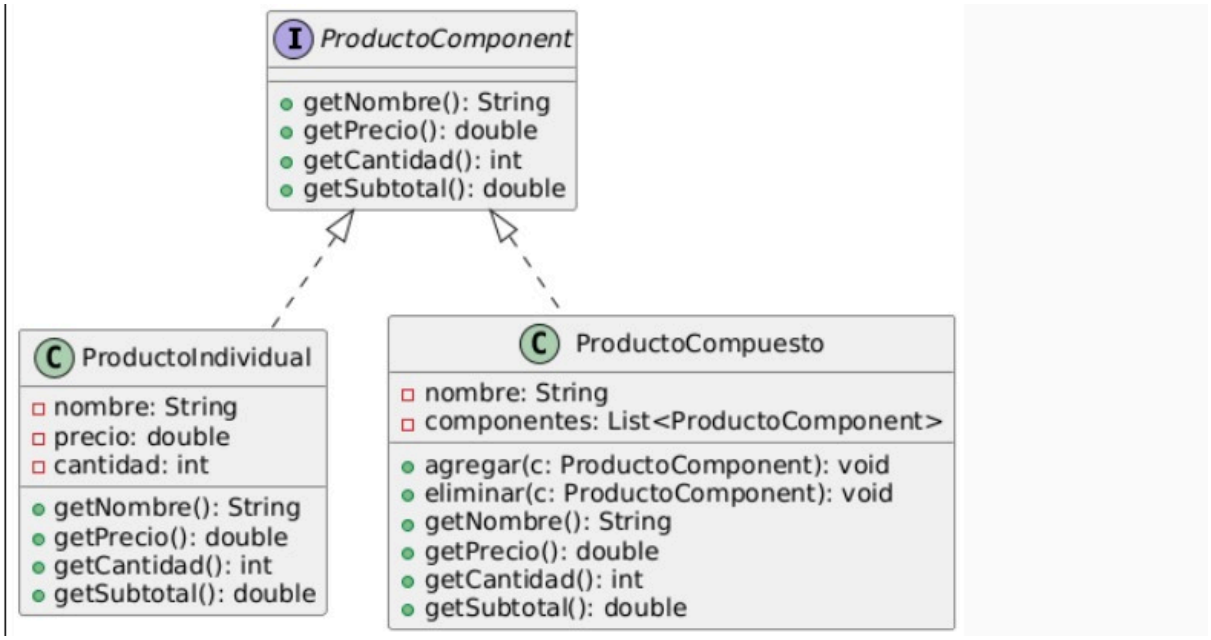
El concepto básico del patrón Composite consiste en representar objetos simples y sus containers (o contenedores, también llamados colecciones en algunos lenguajes, o sea: grupos de objetos) en una clase abstracta de manera que puedan ser tratados uniformemente. Este tipo de estructura se conoce como jerarquía parte-todo (en inglés: part-whole hierarchy), en la que un objeto es siempre, o una parte de un todo, o un todo compuesto por varias partes.

Ventajas y desventajas del patrón de diseño Composite

El patrón Composite es una constante en el desarrollo de software. Los proyectos con estructuras altamente anidadas tienden a beneficiarse de la práctica metodología de los objetos: ya sean objetos simples o complejos, con dependencias simples o complejas.
Pero, a pesar de todas las ventajas, el patrón Composite y su interfaz unificada tienen algunos inconvenientes. La interfaz puede convertirse en un dolor de cabeza para los desarrolladores, ya que la implementación requiere algunas consideraciones nada desdeñables. Por ejemplo, se debe decidir de antemano qué operaciones van a ser definidas en la interfaz y cuáles en las clases Composite.

Ventajas	Desventajas
Facilita la representación de estructuras altamente anidadas	Implementación de interfaz de componentes complicada
Código simple y conciso	Ajustes posteriores de las propiedades Composite complicados
Gran escalabilidad	

Diagrama UML



Código

Interfaz común: ProductoComponent

```
package modelo.composite;
public interface ProductoComponent {
string getNombre();
double getPrecio();
int getCantidad();
double getSubtotal();
}
```

Clase hoja productoIndividual

```
package modelo.composite;


public class ProductoIndividual implements ProductoComponent{
    private String nombre;
    private double precio;
    private int cantidad;

    public ProductoIndividual(String nombre, double precio, int
cantidad) {
        this.nombre = nombre;
        this.precio = precio;
        this.cantidad = cantidad;
```

```
    }

    @Override
    public String getNombre() {
        return nombre;
    }

    @Override
    public double getPrecio() {
        return precio;
    }

    @Override
    public int getCantidad() {
        return cantidad;
    }

    @Override
    public double getSubtotal() {
        return precio * cantidad;
    }
}
```

Clase compuesta productoCompuesto

```
package modelo.composite;
import java.util.ArrayList;
import java.util.List;

public class ProductoCompuesto implements ProductoComponent {
    private String nombre;
    private List<ProductoComponent> componentes = new
ArrayList<>();

    public ProductoCompuesto(String nombre) {
        this.nombre = nombre;
    }

    public void agregar(ProductoComponent componente) {
        componentes.add(componente);
    }
```

```
    public void eliminar(ProductoComponent componente) {
        componentes.remove(componente);
    }

    @Override
    public String getNombre() {
        return nombre;
    }

    @Override
    public double getPrecio() {
        return
componentes.stream().mapToDouble(ProductoComponent::getPrecio).sum
();
    }

    @Override
    public int getCantidad() {
        return componentes.size();
    }

    @Override
    public double getSubtotal() {
        return
componentes.stream().mapToDouble(ProductoComponent::getSubtotal).s
um();
    }
}
```