
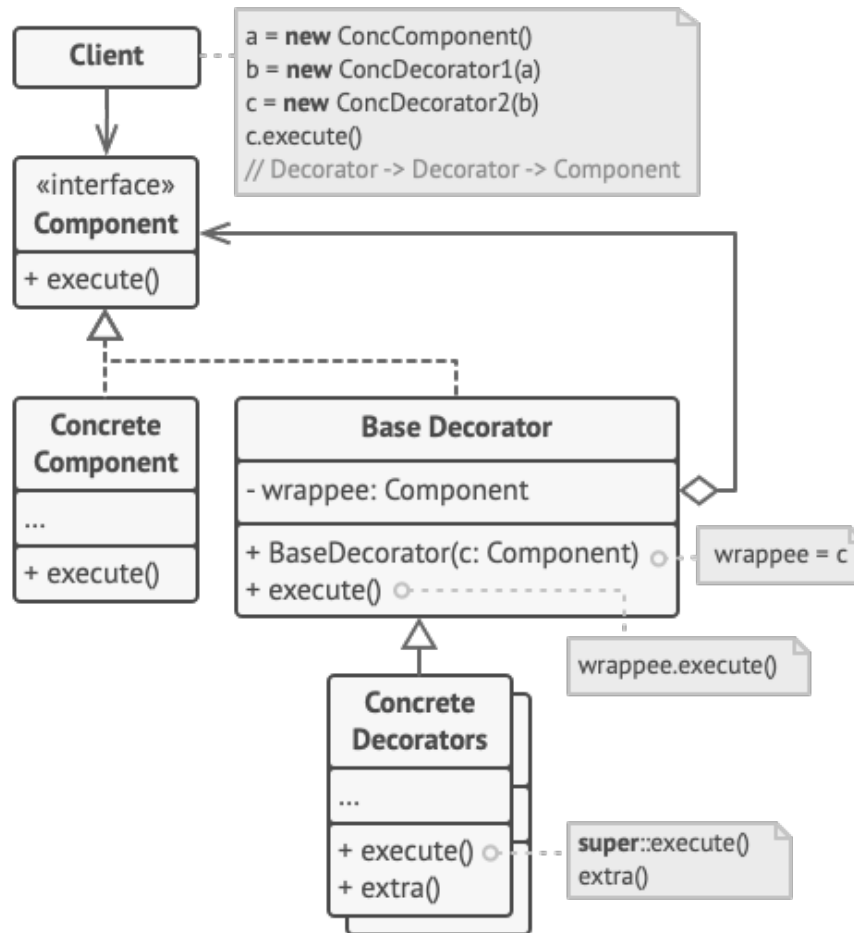
	VICERRECTORADO DOCENTE	Código: GUIA-PRL-001
	CONSEJO ACADÉMICO	Aprobación: 2016/04/06
Formato: Guía de Práctica de Laboratorio / Talleres / Centros de Simulación		

			PRÁCTICA DE LABORATORIO		
CARRERA: COMPUTACIÓN			ASIGNATURA: Programación Aplicada		
NRO. PRÁCTICA:	3	TÍTULO PRÁCTICA: Patrón Decorador (Decorator)			
OBJETIVO ALCANZADO:					
<ul style="list-style-type: none">Identificar los cambios importantes de JavaDiseñar e Implementar las nuevas técnicas de programaciónEntender los patrones de Java					
ACTIVIDADES DESARROLLADAS					
1. Diseñar e implementar un patrón de diseño y verificar su funcionamiento					
Patrón Decorador (Decorator)					
<p>Este patrón de diseño se basa en añadir funcionalidad a un objeto que ya existe sin alterar su estructura. Es un patrón del tipo estructural ya que actúa como un envoltorio (wrapper) de una clase ya existente. Este patrón de diseño utiliza las relaciones de agregación y/o composición para su implementación, con las cuales se pueden agregar funcionalidades de forma dinámica a objetos y clases. Se prefieren estas relaciones por encima de la herencia ya que, en la mayoría de lenguajes de programación, solo se puede tener una relación de herencia, y estas son estáticas, mientras que con las relaciones de composición y agregación uno puede crear diferentes decoradores que tienen el mismo supertipo que el objeto a decorar, pudiendo pasar objetos de las clases decoradoras en lugar de objetos de la clase original y esto nos permite poder decorar a un mismo objeto de forma recursiva.</p>					
Implementación					
<p>Supongamos que tenemos una clase interfaz que nos indica los métodos base para un objeto, a la cual la implementamos mediante una relación de herencia con otra clase concreta. Ahora supongamos que a esta misma clase concreta necesitamos agregar funcionalidades antes o después de creado el objeto, pero no necesitamos cambiar su estructura, solo en forma de añadiduras a la clase. Lo que se hace es crear una clase decoradora base, que heredará la misma clase interfaz, y nos servirá para las demás clases decoradoras, esta clase tendrá como atributo un objeto de la clase envuelta, que puede ser la clase original u otro decorador de la clase original, y entonces se agregarán los métodos o funciones que necesitemos, de esta forma la clase original se mantendrá intacta, y las clases decoradoras irán ejecutando sus métodos de forma recursiva, añadiendo las funcionalidades implementadas sobre la clase original.</p>					



2. Probar y modificar el patrón de diseño a fin de generar cuales son las ventajas y desventajas

Ventajas y Desventajas

Ventajas

- Permite añadir funcionalidades a las clases en tiempo de ejecución.
- Se puede utilizar como una alternativa a las subclases. Estas añaden funcionalidades en tiempo de compilación, sin embargo, se debe tener una subclase para cada tipo de objeto que se podría dar en el programa.
- Ofrece flexibilidad en las funcionalidades, es decir, se las puede ir añadiendo a medida que las necesitamos.

Desventajas

- Se puede llegar a confundir cuando se tienen varios objetos con diferentes decoradores cada uno.
- El utilizar instancias de un mismo objeto nos puede llegar a agrandar el código ya que se necesitan crear e instanciar varios objetos con varios decoradores.
- El añadir varios decoradores a un objeto puede hacernos perder conocimiento de cuantos decoradores hemos añadido a un solo objeto.

3. Realizar práctica codificando los códigos de los patrones y su estructura**Producto**

```
package ups.edu.ec.modelo;
/**
 *
 * @author tano
 */
public class Producto {
    private String descripcion;
    private int cantidad;
    private double precio;

    public Producto() {
    }
    public Producto(String descripcion, int cantidad, double precio) {
        this.descripcion = descripcion;
        this.cantidad = cantidad;
        this.precio = precio;
    }

    public String getDescripcion() {
        return descripcion;
    }

    public void setDescripcion(String descripcion) {
        this.descripcion = descripcion;
    }

    public int getCantidad() {
        return cantidad;
    }

    public void setCantidad(int cantidad) {
        this.cantidad = cantidad;
    }

    public double getPrecio() {
        return precio;
    }

    public void setPrecio(double precio) {
        this.precio = precio;
    }

    @Override
    public String toString() {
        return "Descripción: "+descripcion+" Cantidad: "+cantidad+" Precio: "+precio+" CT: "+
        (precio*cantidad);
    }
}
```

IOrden

```
package ups.edu.ec.modelo;

/**
 *
 * @author tano
 */
public interface IOrden {
    public void agregarProducto(Producto p);
    public double total();
    public void mostrar();
}
```

Orden

```
package ups.edu.ec.modelo;
import java.util.ArrayList;
import java.util.List;
/**
 *
 * @author tano
 */
public class Orden implements IOrden{

    private List<Producto> productos;
    private double total;

    public Orden(int numero) {
        productos = new ArrayList<>();
    }

    public Orden() {
        productos = new ArrayList<>();
    }

    public void agregarProducto(Producto p){
        productos.add(p);
    }

    @Override
    public double total() {
        total = 0;
        productos.stream().forEach(p -> {
            total += (p.getPrecio()*p.getCantidad());
        });
        return total;
    }
}
```

```
@Override
public void mostrar() {
    System.out.println("\nOrden");
    productos.stream().forEach(p -> System.out.println(p.toString()));
    System.out.println("Total: "+total());
}
}
```

DecoradorBase

```
package ups.edu.ec.modelo;

/**
 *
 * @author tano
 */
public abstract class DecoradorBase implements IOrden{
    protected IOrden orden;

    public DecoradorBase(IOrden orden) {
        this.orden = orden;
    }

    public void agregarProducto(Producto p){
        orden.agregarProducto(p);
    }

    @Override
    public double total(){
        return orden.total();
    }

    @Override
    public void mostrar(){
        orden.mostrar();
    }
}
```

DecoradorDescuento

```
package ups.edu.ec.modelo;

/**
 *
 * @author tano
 */
public class DecoradorDescuento extends DecoradorBase{
```

protected double descuento = 0.4;

```
public DecoradorDescuento(IOrden orden) {  
    super(orden);  
}
```

```
public double subTotal(){  
    return orden.total();  
}
```

```
public double getDescuento() {  
    return descuento;  
}
```

```
public void setDescuento(double descuento) {  
    this.descuento = descuento;  
}
```

```
@Override  
public double total(){  
    double total = orden.total()*descuento;  
    total = orden.total()-total;  
    return total;  
}
```

```
@Override  
public void mostrar(){  
    orden.mostrar();  
    System.out.println("Descuento: "+(descuento*100)+"%");  
    System.out.println("Costo Final: "+total());  
}  
}
```

DecoradorDomicilio

```
package ups.edu.ec.modelo;  
/**  
 *  
 * @author tano  
 */  
public class DecoradorDomicilio extends DecoradorBase{  
    protected double costo = 2;  
  
    public DecoradorDomicilio(IOrden orden) {  
        super(orden);  
    }  
  
    @Override  
    public double total(){  
        return orden.total()+costo;  
    }  
}
```

```
@Override
public void mostrar(){
    orden.mostrar();
    System.out.println("Costo de entrega a domicilio: $" + costo);
    System.out.println("Precio con costo de envío: " + total());
}
}
```

PatronDecoradores

```
package ups.edu.ec.test;
import ups.edu.ec.modelo.DecoradorDescuento;
import ups.edu.ec.modelo.DecoradorDomicilio;
import ups.edu.ec.modelo.IOrden;
import ups.edu.ec.modelo.Orden;
import ups.edu.ec.modelo.Producto;

/**
 *
 * @author tano
 */
public class PatronDecoradores {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Producto manzanas = new Producto("manzana", 5, 0.10);
        Producto naranjas = new Producto("naranja", 10, 0.25);
        Producto mangos = new Producto("mango", 7, 0.50);
        IOrden orden = new Orden();
        orden.agregarProducto(mangos);
        orden.agregarProducto(manzanas);
        orden.agregarProducto(naranjas);
        orden.mostrar();
        orden = new DecoradorDescuento(orden);
        orden.mostrar();
        orden = new DecoradorDomicilio(orden);
        orden.mostrar();
    }
}
```

Resultado en consola

```
Orden
Descripción: mango Cantidad: 7 Precio: 0.5 CT: 3.5
Descripción: manzana Cantidad: 5 Precio: 0.1 CT: 0.5
Descripción: naranja Cantidad: 10 Precio: 0.25 CT: 2.5
Total: 6.5

Orden
Descripción: mango Cantidad: 7 Precio: 0.5 CT: 3.5
Descripción: manzana Cantidad: 5 Precio: 0.1 CT: 0.5
Descripción: naranja Cantidad: 10 Precio: 0.25 CT: 2.5
Total: 6.5
Descuento: 40.0%
Costo Final: 3.9

Orden
Descripción: mango Cantidad: 7 Precio: 0.5 CT: 3.5
Descripción: manzana Cantidad: 5 Precio: 0.1 CT: 0.5
Descripción: naranja Cantidad: 10 Precio: 0.25 CT: 2.5
Total: 6.5
Descuento: 40.0%
Costo Final: 3.9
Costo de entrega a domicilio: $2.0
Precio con costo de envío: 5.9
BUILD SUCCESSFUL (total time: 0 seconds)
```

RESULTADO(S) OBTENIDO(S):

- Realizar procesos de investigación sobre los patrones de diseño de Java
- Entender los patrones y su utilización dentro de aplicaciones Java.
- Entender las funcionalidades basadas en patrones.


CONCLUSIONES:

- El Patrón Decorador nos ayuda a no cargar tanto nuestro código al añadir funcionalidades, pero si se utiliza en exceso puede llegar a confundirnos demasiado el código al perder de vista todas las instancias que hemos creado.

RECOMENDACIONES:

- Entender el patrón decorador y cual es su objetivo
- Comprender las relaciones entre clases de agregación y composición
- Interpretar un diagrama de clases e implementarlo en un programa

Nombre de estudiante: Martín Sebastián Toledo Torres

 UNIVERSIDAD POLITÉCNICA SALESIANA ECUADOR	VICERRECTORADO DOCENTE	Código: GUIA-PRL-001
	CONSEJO ACADÉMICO	Aprobación: 2016/04/06
Formato: Guía de Práctica de Laboratorio / Talleres / Centros de Simulación		

Firma de estudiante:

