Taller 5 DPOO
11/05/2022

Actividades:

1. Seleccione un patrón del libro y estúdielo con cuidado. Para mejorar su comprensión, le recomendamos buscar otras descripciones del patrón y especialmente otros nombres para el mismo patrón. Por ejemplo, el patrón Observer aparece muchas veces mencionado como Publisher-Subscriber.
2. Busque un proyecto en algún repositorio de código público que use ese patrón.
3. Reconstruya y estudie el diseño de la parte del proyecto donde se usa el patrón: qué responsabilidades tiene y cuál es el rol del patrón.

Deben entregar un documento que contenga al menos la siguiente información.

- Información general del proyecto, deben incluir la URL para consultar el proyecto.
- Información y estructura del fragmento del proyecto donde aparece el patrón. No se limiten únicamente a los elementos que hacen parte del patrón: para que tenga sentido su uso, probablemente van a tener que incluir elementos cercanos que sirvan para contextualizarlo.
- Información general sobre el patrón: qué patrón es y para qué se usa usualmente
- Información del patrón aplicado al proyecto: explicar cómo se está utilizando el patrón dentro del proyecto
- ¿Por qué tiene sentido haber utilizado el patrón en ese punto del proyecto? ¿Qué ventajas tiene?
- ¿Qué desventajas tiene haber utilizado el patrón en ese punto del proyecto?
- ¿De qué otras formas se le ocurre que se podrían haber solucionado, en este caso particular, los problemas que resuelve el patrón?

Solución:

Decorator (175) Adjunta las responsabilidades adicionales a un objeto dinámicamente. Los decoradores proporcionan una alternativa flexible a la subclasificación para ampliar la funcionalidad.

Estructural: Composite (163) y Decorator (175) son especialmente útiles para construir estructuras complejas en tiempo de ejecución.

USOS:

1. Ampliación de la funcionalidad mediante subclases. A menudo, personalizar un objeto mediante subclases no es fácil. Cada nueva clase tiene una sobrecarga de implementación fija (inicialización, finalización, etc.). Definir una subclase también requiere una comprensión profunda de la clase principal. Por ejemplo, la invalidación de una operación puede requerir la invalidación de otra. Es posible que se requiera

una operación invalidada para llamar a una operación heredada. Y la creación de subclases puede conducir a una explosión de clases, porque es posible que tenga que introducir muchas subclases nuevas incluso para una extensión simple.
La composición de objetos en general y la delegación en particular proporcionan alternativas flexibles a la herencia para combinar el comportamiento. Se puede agregar nueva funcionalidad a una aplicación al componer objetos existentes de nuevas maneras en lugar de definir nuevas subclases de clases existentes. Por otro lado, el uso intensivo de la composición de objetos puede hacer que los diseños sean más difíciles de entender. Muchos patrones de diseño producen diseños en los que puede introducir funciones personalizadas simplemente definiendo una subclase y componiendo sus instancias con las existentes.
Patrones de diseño: Puente (151), Cadena de responsabilidad (223), Compuesto (163), Decorador (175), Observador (293), Estrategia (315).

2. Incapacidad para alterar las clases convenientemente. A veces tienes que modificar una clase que no se puede modificar convenientemente. Tal vez necesites el código fuente y no lo tengas (como puede ser el caso de una biblioteca de clases comercial). O tal vez cualquier cambio requiere modificar muchas subclases existentes.
Los patrones de diseño ofrecen formas de modificar las clases en tales circunstancias.
Patrones de diseño: Adaptador (139), Decorador (175), Visitante (331)

Explicación:

- Es un patrón estructural que nos permite extender funcionalidades de una clase sin tener que usar herencia. No se necesita implementar una jerarquía de clases para cada combinación de posibles objetos.
- Esto es útil ya que mientras en un programa pequeño no presenta mayor problema, en uno más grande puede llegar a presentar problemas por la cantidad de clases existentes: al no saturar el programa con estas clases, el código llegaría a ser más organizado y visible lo que haría que se entendiera mejor.
- Un decorador es un componente al cual se le pueden agregar otros componentes (con diferentes características y comportamiento) que sería la forma de implementar funcionalidades sin utilizar la herencia

- Ejemplo explicativo:

  En videojuegos, un objeto (personaje) que tenga un atributo de armadura tendrá asociada una armadura sabiendo que existen diferentes tipos de armaduras que hacen que sea más pesado, tenga más defensa, se mueva de un modo, etc.., mismo caso si también se le puede agregar cascos o botas, pues el hecho de que un personaje pueda tener botas y no casco, lo hace tener diferente comportamiento a uno que no posea ninguno. Si se organiza con herencia estos objetos, se crean clases para los diferentes personajes que se puedan crear, con el patrón decorator se pueden generar objetos de

manera dinámica. El patrón decorator permite extender comportamientos, un decorador es la armadura, casco y botas. Decoradores se pueden juntar pues las características no son exclusivas, entonces se pueden añadir varios decoradores, es decir, poner casco, armadura y botas. Un decorador es un componente que tiene un componente dentro, por lo cual, se puede hacer un decorador que tiene un decorador adentro. Siguiendo el ejemplo, se puede tener un decorador con armadura, un decorador con casco y un decorador con botas. Dentro de las funciones del decorador se modifica el comportamiento del componente. El orden en el que se añaden los decoradores es importante porque influye en el comportamiento.

```
function algunaFuncionDelJuego() {
  // Algunas cosas por aqui

  //Creamos el enemigo
  let enemigoDesnudo = new ConcreteEnemy();

  //Añadimos el primer decorador, por ejemplo podria ser
  //un DecoradorDeArmadura
  let enemigoConArmadura = new ArmourDecorator(enemigoDesnudo);

  //Añadimos el segundo decorador, por ejemplo podria ser
  //un DecoradorDeCasco
  let enemigoConArmaduraYCasco = new HelmetDecorator(enemigoConArmadura);

  // Algunas cosas por allá
}
```

Repositorio ejemplo: https://github.com/mitocode21/patrones-diseno.git

- Contexto: Cliente se acerca al banco y quiere abrir cuenta de ahorros, allí se le pregunta si quiere un blindaje para su cuenta, entonces se define una clase normal y una clase con blindaje.

- Cliente: Se crea el objeto persona con su información básica como atributos (nombre, id, edad)

- Cuenta: es la clase base que tiene los valores iniciales para abrir una cuenta

```java
package com.mitocode.model;

public class Cuenta {

	private int id;
	private String cliente;

	public Cuenta() {

	}

	public Cuenta(int id, String cliente) {
		this.id = id;
		this.cliente = cliente;
	}

	public int getId() {
		return id;
	}

	public void setId(int id) {
		this.id = id;
	}

	public String getCliente() {
		return cliente;
	}

	public void setCliente(String cliente) {
		this.cliente = cliente;
	}

}
```

- ICuentaBancaria: es una interfaz que abre una cuenta de clase Cuenta. Ayuda a representar clase de Ahorro o Corriente

- CuentaAhorro: implementa la interfaz

- CuentaCorriente: implementa la interfaz

- CuentaDecorador: es una clase abstracta que tiene patrón de decorador que recibe cualquier implementación de la interfaz (ya sea CuentaAhorro o CuentaCorriente)

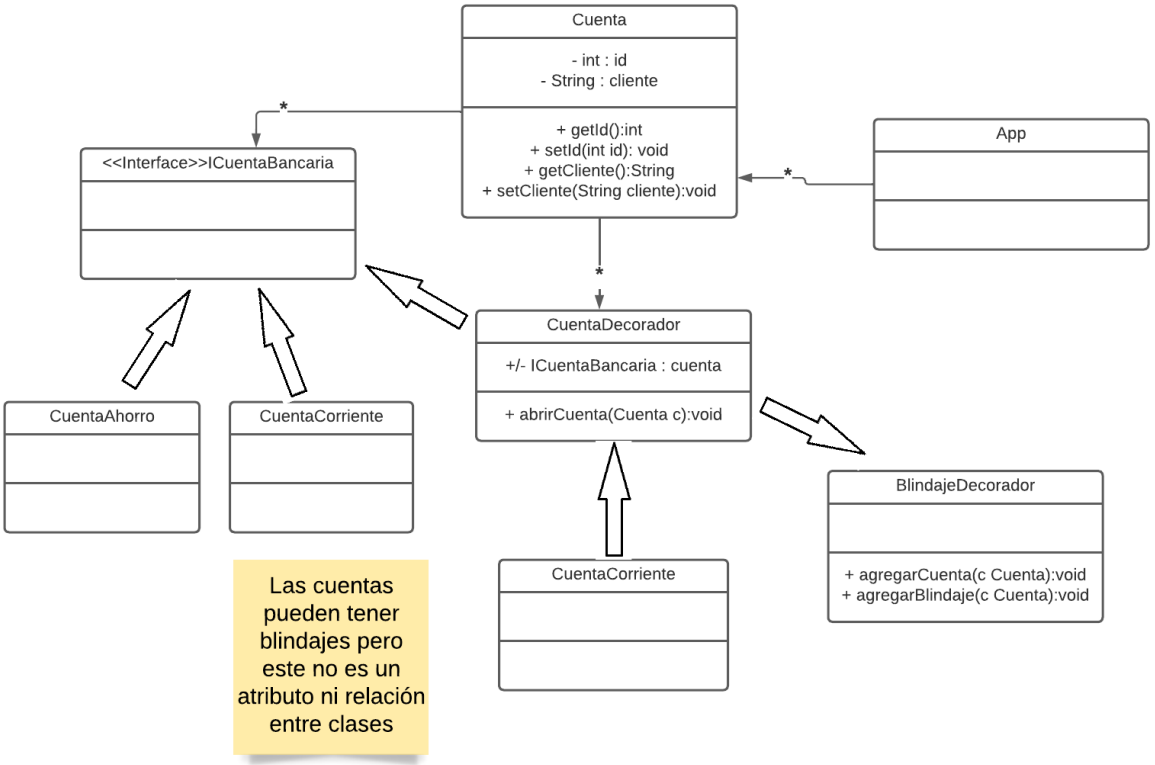
- BlindajeDecorador: extiende el decorador base, es decir, CuentaDecorador que recibe la instancia de la interfaz (ya sea CuentaAhorro o CuentaCorriente). En esta clase se le agregan metodos que representan las funcionalidades adicionales.

```java
package com.mitocode.decorador;

import com.mitocode.interf.ICuentaBancaria;
import com.mitocode.model.Cuenta;

public class BlindajeDecorador extends CuentaDecorador {

	public BlindajeDecorador(ICuentaBancaria cuentaDecorada) {
		super(cuentaDecorada);
	}

	@Override
	public void abrirCuenta(Cuenta c) {
		cuentaDecorada.abrirCuenta(c);
		agregarBlindaje(c);
	}

	public void agregarBlindaje(Cuenta c) {
		System.out.println("Se agregó blindaje a la cuenta del cliente " + c.getCliente());
	}

}
```

- Ventajas de decorador: Si se quiere agregar un blindaje nuevo u otro comportamiento, no se debería cambiar la estructura de la clase Cuenta o la clase CuentaAhorro para tener nuevas clases como: CuentaAhorroBlindaje y CuentaCorrienteBlindaje y de allí, establecer un comportamiento diferente, ya sea en el acceso a la cuenta o al ejecutar pagos o transacciones. Simplemente se crear decoradores que permitan adquirir nuevas características y allí podrán crearse nuevos comportamientos asociados a la cuenta (métodos)

- Desventajas de decorador: Puede que requiera mayor conocimiento de las funcionalidades del código, pues al tener decoradores en decoradores, el orden importa, pues una cuenta debe ser primero CuentaBlindada antes que tener un BlindajeDecorador, pues si fuera al revés, habría problemas en el comportamiento del programa, siendo una instancia creada sin blindaje.

Diagrama de clases:



Otra solución al planteamiento: Si no se usará el patrón de decorador, se puede buscar que cada cuenta que extiende la interfaz tenga un atributo y a partir de este se ejecutan diferentes métodos que corresponden al comportamiento. Por ejemplo, los métodos podrían evaluar si la entrada es “blinded” o “noblinded” y a partir de ello, ejecutarán una u otra instrucción. Incluso se puede usar un atributo bool llamado blinded y partir de allí.

Referencias:

E. Sciore, Java Program Design, Principles, Polymorphism, and Patterns, Apress, 2019.

Info: https://www.tutorialspoint.com/design_pattern/decorator_pattern.htm

Repositorio ejemplo: https://github.com/nipafx/demo-decorator-java-8.git muy compleja

Repositorio ejemplo: https://github.com/mitocode21/patrones-diseno.git suave

Integrantes:

- j.serratos
- da.acostac1
- c.caro