Patrones Estructurales

Los patrones estructurales se centran en la forma en que las clases y los objetos están organizados o estructurados para formar sistemas más grandes. Su principal objetivo es facilitar el diseño y la creación de estructuras de datos y objetos complejas, asegurando que estas estructuras sean flexibles y eficientes.

Características de los Patrones Estructurales:

1. **Composición de Objetos:** Los patrones estructurales a menudo usan la composición en lugar de la herencia. Esto significa que en lugar de crear nuevas clases derivadas, los objetos existentes se combinan para crear nuevas funcionalidades.
2. **Adaptabilidad:** Facilitan la adaptación de interfaces existentes para que diferentes sistemas puedan trabajar juntos de manera efectiva.
3. **Optimización del Código:** Ayudan a reducir la complejidad del código y a optimizar las interacciones entre objetos, lo que puede mejorar el rendimiento y la mantenibilidad.

Ejemplos Comunes de Patrones Estructurales:

1. **Adaptador (Adapter):** Convierte la interfaz de una clase en otra interfaz que un cliente espera. Permite que clases que no podrían trabajar juntas debido a interfaces incompatibles lo hagan.
2. **Decorador (Decorator):** Permite añadir responsabilidades adicionales a un objeto de manera dinámica. Los decoradores proporcionan una alternativa flexible a la subclasificación para extender la funcionalidad.
3. **Fachada (Facade):** Proporciona una interfaz simplificada para un conjunto de interfaces en un subsistema. Es útil para estructurar sistemas complejos y hacerlos más fáciles de usar.
4. **Proxy:** Proporciona un sustituto o marcador de posición para controlar el acceso a un objeto. Puede ser utilizado para retardar la creación del objeto real, controlar el acceso a él, o añadir funcionalidad adicional.

```java
// Interface Componente
interface Cafeteria {
    String servir();
    double precio();
}

// Implementación concreta del componente
class CafeSimple implements Cafeteria {
    @Override
    public String servir() {
        return "Café";
    }

    @Override
    public double precio() {
        return 10.0;
    }
}

// Decorador abstracto
abstract class CafeDecorador implements Cafeteria {
    protected Cafeteria cafeteria;

    public CafeDecorador(Cafeteria cafeteria) {
        this.cafeteria = cafeteria;
    }
```

```java
    @Override
    public String servir() {
        return cafeteria.servir();
    }

    @Override
    public double precio() {
        return cafeteria.precio();
    }
}

// Decoradores concretos
class Leche extends CafeDecorador {
    public Leche(Cafeteria cafeteria) {
        super(cafeteria);
    }

    @Override
    public String servir() {
        return cafeteria.servir() + " con leche";
    }

    @Override
    public double precio() {
        return cafeteria.precio() + 2.0;
    }
}
```

```java
class Chocolate extends CafeDecorador {
    public Chocolate(Cafeteria cafeteria) {
        super(cafeteria);
    }

    @Override
    public String servir() {
        return cafeteria.servir() + " con chocolate";
    }

    @Override
    public double precio() {
        return cafeteria.precio() + 3.0;
    }
}

// Uso del patrón Decorador
public class Main {
    public static void main(String[] args) {
        Cafeteria cafeSimple = new CafeSimple();
        System.out.println(cafeSimple.servir() + " - Precio: " + cafeSimple.precio());

        Cafeteria cafeConLeche = new Leche(cafeSimple);
        System.out.println(cafeConLeche.servir() + " - Precio: " + cafeConLeche.precio());

        Cafeteria cafeConLecheYChocolate = new Chocolate(cafeConLeche);
        System.out.println(cafeConLecheYChocolate.servir() + " - Precio: " + cafeConLecheYChocolate.precio());
    }
}
```

Explicación:

Aquí, CafeSimple representa un café básico. Los decoradores Leche y Chocolate añaden características adicionales (leche y chocolate) al café de manera dinámica sin modificar la clase original.