Decorator Pattern

Introducción y nombre

Decorator, Estructural. Añade dinámicamente funcionalidad a un objeto.

Intención

El patrón decorator permite añadir responsabilidades a objetos concretos de forma dinámica. Los decoradores ofrecen una alternativa más flexible que la herencia para extender las funcionalidades.

También conocido como

Decorador, Wrapper (igual que el patrón Adapter).

Motivación

A veces se desea adicionar responsabilidades a un objeto pero no a toda la clase. Las responsabilidades se pueden adicionar por medio de los mecanismos de Herencia, pero este mecanismo no es flexible porque la responsabilidad es adicionada estáticamente. La solución flexible es la de rodear el objeto con otro objeto que es el que adiciona la nueva responsabilidad. Este nuevo objeto es el Decorator.

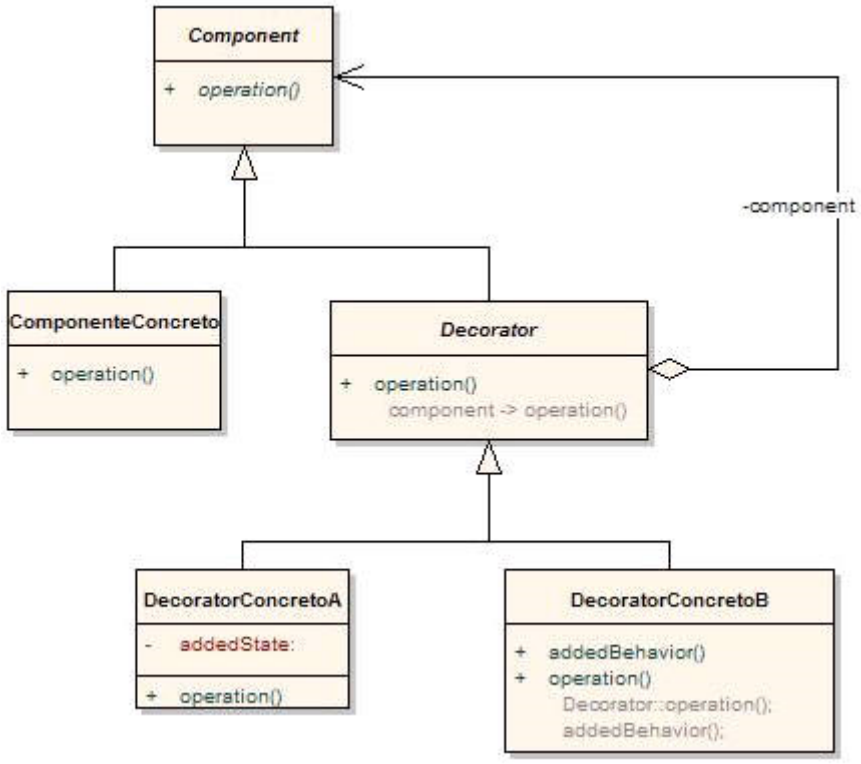
Solución

Este patrón se debe utilizar cuando:

Hay una necesidad de extender la funcionalidad de una clase, pero no hay razones para extenderlo a través de la herencia.

Se quiere agregar o quitar dinámicamente la funcionalidad de un objeto.

Diagrama UML



Participantes

Component: define la interface de los objetos a los que se les pueden adicionar responsabilidades dinámicamente.

ComponenteConcreteo: define el objeto al que se le puede adicionar una responsabilidad.

Decorator: mantiene una referencia al objeto Component y define una interface de acuerdo con la interface de Component.

DecoratorConcreto: adiciona la responsabilidad al Component.

Colaboraciones

Decorator propaga los mensajes a su objeto Component. Opcionalmente puede realizar operaciones antes y después de enviar el mensaje.

Consecuencias

Es más flexible que la herencia: utilizando diferentes combinaciones de unos pocos tipos distintos de objetos decorator, se puede crear muchas combinaciones distintas de comportamientos. Para crear esos diferentes tipos de comportamiento con la herencia se requiere que definas muchas clases distintas.

Evita que las clases altas de la jerarquía estén demasiado cargadas de funcionalidad.

Un componente y su decorador no son el mismo objeto.

Provoca la creación de muchos objetos pequeños encadenados, lo que puede llegar a complicar la depuración.

La flexibilidad de los objetos decorator los hace más propenso a errores que la herencia. Por ejemplo, es posible combinar objetos decorator de diferentes formas que no funcionen, o crear referencias circulares entre los objetos decorator.

Implementación

La mayoría de las implementaciones del patrón Decorator son sencillas. Veamos algunas de las implementaciones más comunes:

Si solamente hay una clase ComponenteConcreto y ninguna clase Component, entonces la clase Decorator es normalmente una subclase de la clase ComponenteConcreto.

A menudo el patrón Decorator es utilizado para delegar a un único objeto. En este caso, no hay necesidad de tener la clase Decorator (abstracto) para mantener una colección de referencias. Sólo conservando una única referencia es suficiente.

Por otro lado, se debe tener en cuenta que un decorador y su componente deben compartir la misma interfaz.

Los componentes deben ser clases con una interfaz sencilla.

Muchas veces se el decorator cabeza de jerarquía puede incluir alguna funcionalidad por defecto.

Código de muestra

Imaginemos que vendemos automóviles y el cliente puede opcionalmente adicionar ciertos componentes (aire acondicionado, mp3 player, etc). Por cada componente que se adiciona, el precio varía.

[code]

```
public interface Vendible {
    public String getDescripcion();

    public int getPrecio();
}

public abstract class Auto implements Vendible {
}


public class FiatUno extends Auto {

    @Override
    public String getDescripcion() {
        return "Fiat Uno modelo 2006";
    }

    @Override
    public int getPrecio() {
        return 15000;
    }
}

public class FordFiesta extends Auto {
    @Override
    public String getDescripcion() {
        return "Ford Fiesta modelo 2008";
    }

    @Override
    public int getPrecio() {
        return 25000;
    }
}

public abstract class AutoDecorator implements Vendible {
    private Vendible vendible;

    public AutoDecorator(Vendible vendible) {
        this.vendible = vendible;
    }
```

```
    public Vendible getVendible() {
        return vendible;
    }

    public void setVendible(Vendible vendible) {
        this.vendible = vendible;
    }
}

public class CdPlayer extends AutoDecorator {

    public CdPlayer(Vendible vendible) {
        super(vendible);
    }

    @Override
    public String getDescripcion() {
        return getVendible().getDescripcion() + " + CD Player";
    }

    @Override
    public int getPrecio() {
        return getVendible().getPrecio() + 100;
    }
}

public class AireAcondicionado extends AutoDecorator {

    public AireAcondicionado(Vendible vendible) {
        super(vendible);
    }

    @Override
    public String getDescripcion() {
        return getVendible().getDescripcion() + " + Aire Acondicionado";
    }

    @Override
    public int getPrecio() {
        return getVendible().getPrecio() + 1500;
    }
}

public class Mp3Player extends AutoDecorator {

    public Mp3Player(Vendible vendible) {
        super(vendible);
    }

    @Override
    public String getDescripcion() {
        return getVendible().getDescripcion() + " + MP3 Player";
    }

    @Override
    public int getPrecio() {
        return getVendible().getPrecio() + 250;
    }
}

public class Gasoil extends AutoDecorator {

    public Gasoil(Vendible vendible) {
        super(vendible);
    }
```

```
    @Override
    public String getDescripcion() {
        return getVendible().getDescripcion() + " + Gasoil";
    }

    @Override
    public int getPrecio() {
        return getVendible().getPrecio() + 1200;
    }
}
```

Probemos el funcionamiento del ejemplo:

```
        public static void main(String[] args) {
                Vendible auto = new FiatUno();
                auto = new CdPlayer(auto);
                auto = new Gasoil(auto);

                System.out.println(auto.getDescripcion());
                System.out.println("Su precio es: "+ auto.getPrecio());

                Vendible auto2 = new FordFiesta();
                auto2 = new Mp3Player(auto2);
                auto2 = new Gasoil(auto2);
                auto2 = new AireAcondicionado(auto2);

                System.out.println(auto2.getDescripcion());
                System.out.println("Su precio es: "+ auto2.getPrecio());
        }
```

La salida por consola es:
Fiat Uno modelo 2006 + CD Player + Gasoil
Su precio es: 16300
Ford Fiesta modelo 2008 + MP3 Player + Gasoil + Aire Acondicionado
Su precio es: 27950

[/code]

Cuándo utilizarlo

Dado que este patrón decora un objeto y le agrega funcionalidad, suele ser muy utilizado para adicionar opciones de "embellecimiento" en las interfaces al usuario. Este patrón debe ser utilizado cuando la herencia de clases no es viable o no es útil para agregar funcionalidad. Imaginemos que vamos a comprar una PC de escritorio. Una estándar tiene un precio determinado. Pero si le agregamos otros componentes, por ejemplo, un lector de CD, el precio varía. Si le agregamos un monitor LCD, seguramente también varía el precio. Y con cada componente adicional que le agreguemos al estándar, seguramente el precio cambiará. Este caso, es un caso típico para utilizar el Decorator.

Facade Pattern

Introducción y nombre

Facade, Estructural. Busca simplificar el sistema, desde el punto de vista del cliente.

Intención

Su intención es proporcionar una interfaz unificada para un conjunto de subsistemas, definiendo una interfaz de nivel más alto. Esto hace que el sistema sea más fácil de usar.

También conocido como

Fachada.

Motivación

Este patrón busca reducir al mínimo la comunicación y dependencias entre subsistemas. Para ello, utilizaremos una fachada, simplificando la complejidad al cliente. El cliente debería acceder a un subsistema a través del Facade. De esta manera, se estructura un entorno de programación más sencillo, al menos desde el punto de vista del cliente (por ello se llama "fachada").

Solución

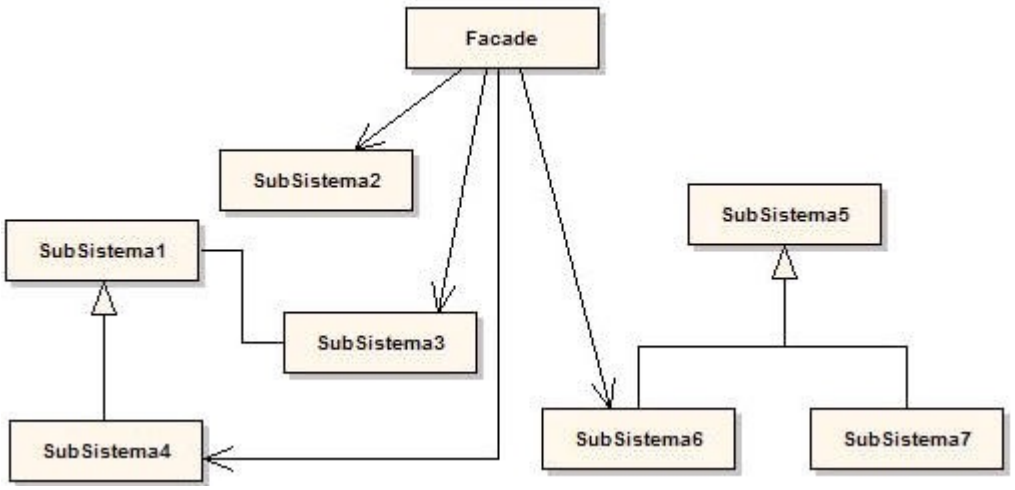
Este patrón se debe utilizar cuando:

Se quiera proporcionar una interfaz sencilla para un subsistema complejo.

Se quiera desacoplar un subsistema de sus clientes y de otros subsistemas, haciéndolo más independiente y portable.

Se quiera dividir los sistemas en niveles: las fachadas serían el punto de entrada a cada nivel. Facade puede ser utilizado a nivel aplicación.

Diagrama UML



Participantes

Facade:

Conoce cuales clases del subsistema son responsables de una petición.

Delega las peticiones de los clientes en los objetos del subsistema.

Subsistema:

Implementar la funcionalidad del subsistema.

Manejar el trabajo asignado por el objeto Facade.

No tienen ningún conocimiento del Facade (no guardan referencia de éste).

Colaboraciones:

Los clientes se comunican con el subsistema a través de la facade, que reenvía las peticiones a los objetos del subsistema apropiados y puede realizar también algún trabajo de traducción

Los clientes que usan la facade no necesitan acceder directamente a los objetos del sistema.

Consecuencias

Oculta a los clientes de la complejidad del subsistema y lo hace más fácil de usar.

Favorece un acoplamiento débil entre el subsistema y sus clientes, consiguiendo que los cambios de las clases del sistema sean transparentes a los clientes.

Facilita la división en capas y reduce dependencias de compilación.

No se impide el acceso a las clases del sistema.

Implementación

Es un patrón muy sencillo de utilizar. Se debe tener en cuenta que no siempre es tan sólo un "pasamanos". Si fuese necesario que el Facade realice una tarea específica antes de devolver una respuesta al cliente, podría hacerlo sin problema.

Lo más importante de todo es que este patrón se debe aplicar en las clases más representativas y no en las específicas. De no ser así, posiblemente no se tenga el nivel alto deseado.

Por aplicación, es ideal construir no demasiados objetos Facade. Sólo algunos representativos que contengan la mayoría de las operaciones básicas de un sistema.

Código de muestra

Imaginemos que estamos, con un equipo de desarrollo, realizando el software para una inmobiliaria. Obviamente una inmobiliaria realiza muchos trabajos diferentes, como el cobro de alquiler, muestra de inmuebles, administración de consorcios, contratos de ventas, contratos de alquiler, etc.

Por una cuestión de seguir el paradigma de programación orientada a objetos, es probable que no se realice todo a una misma clase, sino que se dividen las responsabilidades en diferentes clases.

[code]

```
public class Persona {
}
public class Cliente extends Persona {
}
public class Interesado extends Persona {
}
public class Propietario extends Persona {
}

public class AdministracionAlquiler {
    public void cobro(double monto) {
        // Algoritmo
    }
}
```

```
public class CuentasAPagar {
    public void pagoPropietario(double monto) {
        // Algoritmo
    }
}

public class MuestraPropiedad {
    public void mostraPropiedad(int numeroPropiedad) {
        // Algoritmo
    }
}

public class VentaInmueble {
    public void gestionaVenta() {
        // Algoritmo
    }
}

public class Inmobiliaria {

    private MuestraPropiedad muestraPropiedad;
    private VentaInmueble venta;
    private CuentasAPagar cuentasAPagar;
    private AdministracionAlquiler alquiler;

    public Inmobiliaria() {
        muestraPropiedad = new MuestraPropiedad();
        venta = new VentaInmueble();
        cuentasAPagar = new CuentasAPagar();
        alquiler = new AdministracionAlquiler();
    }

    public void atencionCliente(Cliente c) {
        System.out.println("Atendiendo a un cliente");
    }

    public void atencionPropietario(Propietario p) {
        System.out.println("Atendiendo a un propietario");
    }

    public void atencionInteresado(Interesado i) {
        System.out.println("Atencion a un interesado en una propiedad");
    }

    public void atencion(Persona p) {
        if (p instanceof Cliente) {
            atencionCliente((Cliente) p);
        } else if (p instanceof Propietario) {
            atencionPropietario((Propietario) p);
        } else {
            atencionInteresado((Interesado) p);
        }
    }

    public void mostraPropiedad(int numeroPropiedad) {
        muestraPropiedad.mostraPropiedad(numeroPropiedad);
    }

    public void gestionaVenta() {
        venta.gestionaVenta();
    }

    public void paga(int monto) {
        cuentasAPagar.pagoPropietario(monto);
    }
```

```
    public void cobraAlquiler(double monto) {
        alquiler.cobro(monto);
    }
}
```

Probemos como es el funcionamiento del Facade:

```
public class Main {

    public static void main(String[] args) {
        Cliente c = new Cliente();
        Interesado i = new Interesado();
        Inmobiliaria inmo = new Inmobiliaria();
        inmo.atencionCliente(c);
        inmo.atencionInteresado(i);

        MuestraPropiedad muestraPropiedad = new MuestraPropiedad();
        muestraPropiedad.mostraPropiedad(123);
        VentaInmueble venta = new VentaInmueble();
        venta.gestionaVenta();
        AdministracionAlquiler alquiler = new AdministracionAlquiler();
        alquiler.cobro(1200);
        CuentasAPagar cuentasAPagar = new CuentasAPagar();
        cuentasAPagar.pagoPropietario(1100);

        // Lo mismo pero despues del Facade
        Inmobiliaria inmo2 = new Inmobiliaria();
        inmo2.atencion(i);
        inmo2.atencion(c);
        inmo2.mostraPropiedad(123);
        inmo2.gestionaVenta();
        inmo2.cobraAlquiler(1200);
        inmo2.paga(1100);

    }
}
```
[/code]

Cuándo utilizarlo

Sabemos que el Facade busca reducir la complejidad de un sistema. Esto mismo ocurre en ciertos lugares donde tendremos muchas opciones: imaginemos a un banco o edificio público. Es un lugar donde cada persona hace trámites distintos y, por ende, cada persona se encuentra con complicaciones de distinta índole. Es complicado para las personas que trabajan en dichos lugares, por ello es que tienen un especialista por cada tema. Por ello, es muy raro que el cajero sea la misma persona que gestiona un préstamo hipotecario.

Esta misma complejidad se traspasa para el cliente: cuando entramos a un lugar grande con muchas ventanillas, es posible que hagamos la fila en el lugar incorrecto.

¿Cómo se soluciona este caos? Colocando un mostrador de información en la entrada para que todos los clientes vayan directamente al mostrador y allí se va direccionando a las personas al lugar correcto.

A grandes rasgos, se podría decir que el mostrador de información cumple un rol similar a un Facade: todos los clientes se dirigen allí y el se encarga de solucionarnos el problema. En realidad, sabe quién es la persona que lo va a solucionar.

En los proyectos grandes suele ocurrir que se pierde el control de la cantidad de clases y cuando este ocurre, no es bueno obligar a todos los clientes a conocer los subsistemas. Este caso, es un caso ideal para aplicar un Facade.