Decorator

migranitodejava.blogspot.com/2011/06/decorator.html

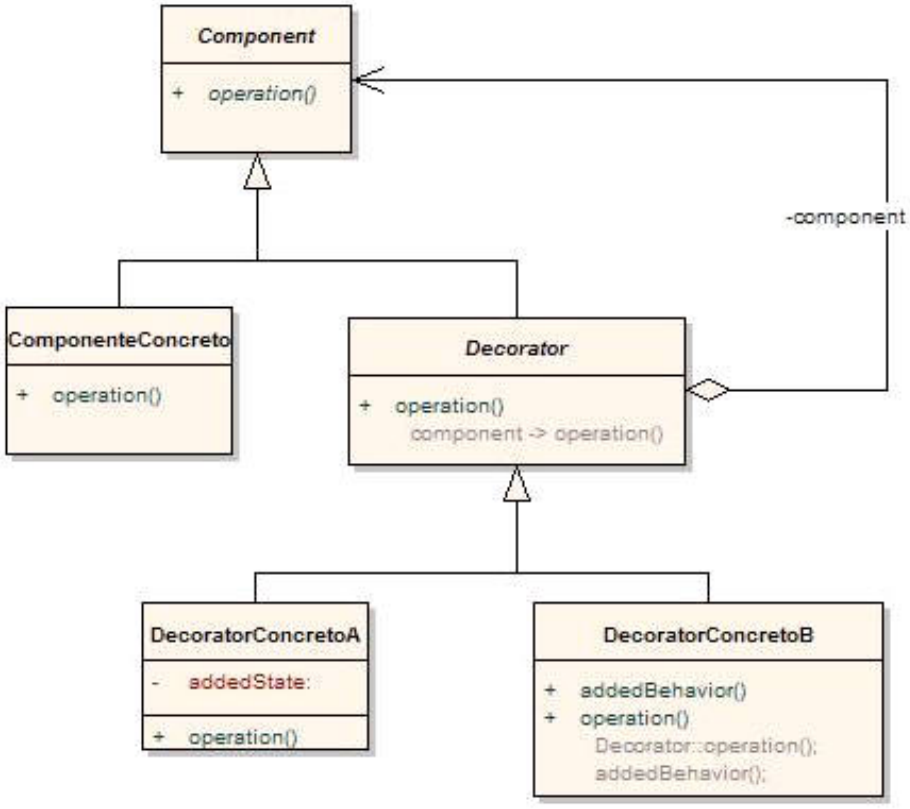
El patrón decorator permite añadir responsabilidades a objetos concretos de forma dinámica. Los decoradores ofrecen una alternativa más flexible que la herencia para extender las funcionalidades.
Es conocido como Wrapper (igual que el patrón Adapter).
A veces se desea adicionar responsabilidades a un 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.

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.

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.

Diagrama UML



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.

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

Ejemplo

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.

```
public interface Vendible {

    public String getDescripcion();
    public int getPrecio();

}
```

```
public abstract class Auto implements Vendible{
// aca irían todos los atributos propios
// de un auto pero no nos interesan para el ejemplo
}
```

Bien, hasta aquí clases comunes de negoci: una interface que implementa la clase Auto y dos tipos de Auto (Ford y Fiat). Ahora veremos en que consiste el Decorator y los decoradores:

```
public class FiatUno extends Auto {

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

    public int getPrecio() {
        return 15000;
    }

}

public class FordFiesta extends Auto {

    public String getDescripcion() {
        return "Ford Fiesta modelo 2008";
    }

    public int getPrecio() {
        return 25000;
    }

}
```

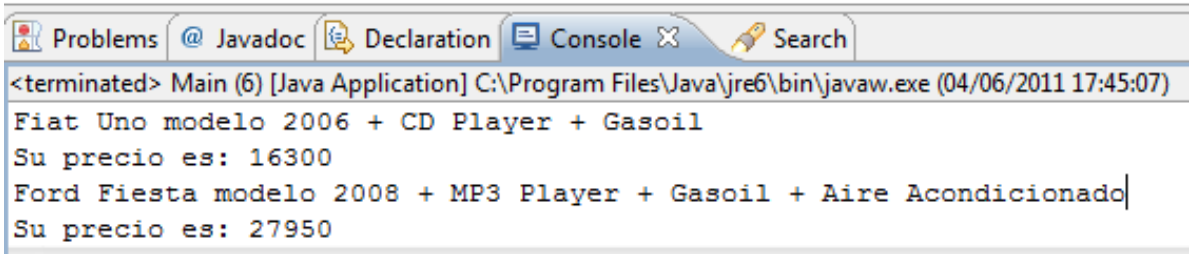
```
public abstract class AutoDecorator implements Vendible {
    private Vendible vendible;

    public AutoDecorator(Vendible vendible) {
        setVendible(vendible);
    }

    public Vendible getVendible() {
        return vendible;
    }

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

}
```

```
public class AireAcondicionado extends AutoDecorator {

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

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

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

}
```

```
public class CdPlayer extends AutoDecorator {

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

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

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

}
  public class Gasoil extends AutoDecorator {

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

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

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

  }
public class Mp3Player extends AutoDecorator {

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

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

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

}
```

Veremos como funciona desde el punto de vista del cliente:

```
    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());

    }
```



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.