Patrón de Diseño Decorator en Java

| *Gustavo*

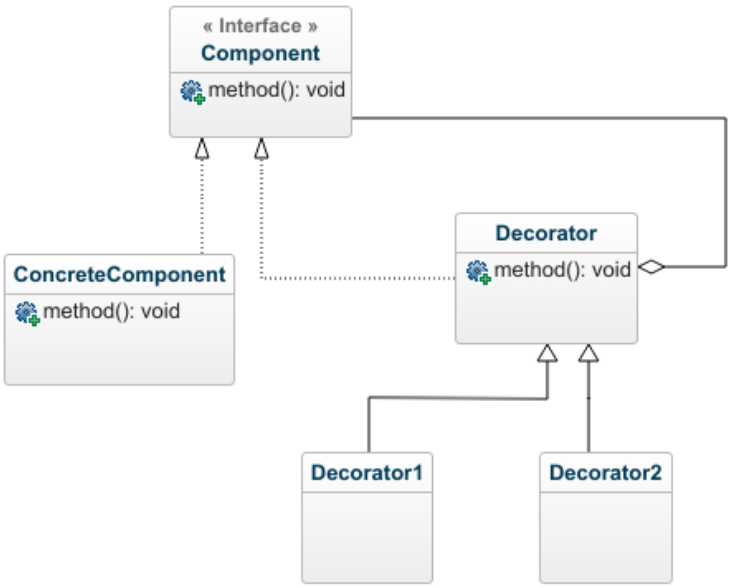
El patrón Decorator en Java permite agregar nuevas funcionalidades a las clases sin modificar su estructura.

El concepto de este patrón es agregar de forma **dinámica** nuevo comportamiento o funcionalidades a la clase principal.

La definición oficial dice que “permite el ajuste dinámico de objetos para modificar sus responsabilidades y comportamientos existentes.”

Cuales son las partes del patrón de diseño Decorator

El patrón decorador se compone principalmente de una Interfaz de la cual se implementa la clase concreta y los decoradores que añadirá mayor funcionalidad a la clase concreta.



- Component Interface: es la interface (puede ser una clase abstracta también) que define la funcionalidad y de la cuál se hereda la clase concreta y los decoradores.
- Concrete Component: es la implementación principal y cuya clase recibirá los decoradores para agregar funcionalidad extra dinámicamente.
- Decorator: puede ser una clase abstracta o no que define el Decorador que hereda de la interfaz Component y de la cual luego se crearán todos los demás decoradores. El decorador debe mantener la referencia al objeto original a fin de invocarlo y luego agregarle otras funcionalidades propias del decorador. Cada decorador tiene una relación con el componente de tipo HAS-A (tiene un).
- Concrete Decorator: son las clases que extienden o implementan el Decorator con la funcionalidad acotada.

Como se crea el patrón de diseño Decorator en Java

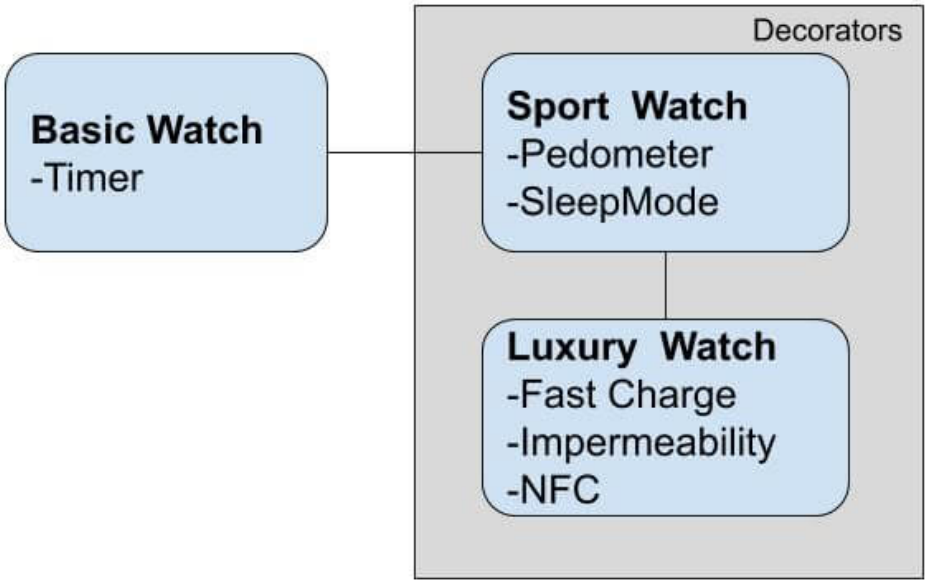
Dijimos que el concepto del patrón decorador es agregar funcionalidades al objeto principal de forma dinámica. Evitando de esta manera la necesidad de crear subclases a la clase principal para agregar funcionalidad.

La ventaja es que de este modo no afectamos a todas las clases. Cada clase define una funcionalidad específica que se agrega y **decora** a la clase principal sin necesidad de crear subclases.

Lo primero que necesitamos buscar el el comportamiento común que tienen todos los objetos y que podría extraerse a una interfaz. Esta interfaz será el contrato que todas las implementaciones, tanto de la clase concreta como la de los decoradores deberá cumplir.

Con un ejemplo se ve mas simple.

Pensemos en un Reloj al cual se le añaden funcionalidades y según las funcionalidades añadidas se convierte en un reloj deportivo o un reloj de Lujo.



Definamos el componente que será la interfaz Watch.

```
package patterns.decorator;

public interface Watch {

    void createFunctionality();
}
```

Ahora creamos la clase concreta BasicWatch a partir de la interfaz

```
package patterns.decorator;

public class BasicWatch implements Watch {

    @Override
    public void createFunctionality() {
        System.out.println(" Basic Watch with: ");
        this.addTimer();
    }

    private void addTimer() {
        System.out.print(" Timer");
    }
}
```

Creamos el decorador WatchDecorator también a partir de la interfaz.

```
package patterns.decorator;

public abstract class WatchDecorator implements Watch {

    private final Watch watch;

    public WatchDecorator(Watch watch) {
        this.watch = watch;
    }
    @Override
    public void createFunctionality() {
        this.watch.createFunctionality();
    }
}
```

Creamos el resto de decoradores a partir del decorador principal añadirán las funcionalidades particulares.

```
package patterns.decorator;

public class SportWatchDecorator extends WatchDecorator {

    public SportWatchDecorator(Watch watch) {
        super(watch);
    }

    @Override
    public void createFunctionality(){
        super.createFunctionality();
        System.out.print(" and more features (Sport Watch): ");
        this.addPedometer();
        this.addSleepMode();
    }

    private void addPedometer() {
        System.out.print(" Pedometer");
    }

    private void addSleepMode() {
        System.out.print(" SleepMode ");
    }
}
```

```
package patterns.decorator;

public class LuxuryWatchDecorator extends WatchDecorator {

    public LuxuryWatchDecorator(Watch watch) {
        super(watch);
    }

    @Override
    public void createFunctionality() {
        super.createFunctionality();
        System.out.print(" and  more features (Luxury Watch): ");
        this.addFastCharge();
        this.addImpermeability();
        this.addNFC();
    }

    private void addFastCharge() {
        System.out.print(" FastCharge ");
    }

    private void addImpermeability() {
        System.out.print(" Impermeability ");
    }

    private void addNFC() {
        System.out.print(" NFC ");
    }
}
```

Bien ahora probemos nuestro patrón decorator.

```
package patterns.decorator;

public class ClientDecoratorPattern {

    public static void main(String... args) {

        Watch basicWatch = new BasicWatch();
        basicWatch.createFunctionality();
        System.out.println("\n---------");

        Watch sportsWatch = new SportWatchDecorator(new BasicWatch());
        sportsWatch.createFunctionality();
        System.out.println("\n---------");

        Watch sportsLuxuryWatch = new LuxuryWatchDecorator(new SportWatchDecorator(new BasicWatch()));
        sportsLuxuryWatch.createFunctionality();
    }
}
```

Que hicimos en el código previo:

- Creamos nuestro objeto concreto y principal BasicWatch que no tiene mucha funcionalidad más que el timer.
- Creamos un objeto decorador SportWatchDecorator que añade mayores funcionalidades al al objeto principal BasicWatch
- Creamos un objeto decorador LuxuryWatchDecorator que añade mayor funcionalidad al decorador SportWatch que a su vez añade funcionalidad al objeto BasicWatch.

La salida de esto es

```
Basic Watch with:
 Timer
---------
 Basic Watch with:
 Timer and more features (Sport Watch):  Pedometer SleepMode
---------
 Basic Watch with:
 Timer and more features (Sport Watch):  Pedometer SleepMode  and  more features (Luxury Watch):  FastCharge  Impe
rmeability  NFC
```

Conclusión

Vimos cómo crear un patrón decorator que nos permite agregar nueva funcionalidad en los casos en que extender la clase principal no resulta una buena opción.

Esta funcionalidad podemos añadirla de forma estática o de forma dinámica según condiciones y nos ayuda a acotar en las clases decorator las características, por lo resulta más simple su implementación.

Puedes descargar este código en github (https://github.com/gustavopeiretti/java-examples) o en gitlab (https://gitlab.com/gustavopeiretti/java-examples)

> Hola! Si mis post te son útiles y te ayudan a aprender algo de ellos, considera brindar tu soporte invitándome un rico café. :) Muchas gracias!
>
> Buy me a coffee (http://buymeacoffee.com/N6toizJlC)

Tags: #Java (https://gustavopeiretti.com/tags/java/) #pattern (https://gustavopeiretti.com/tags/pattern/)

← **ARTÍCULO ANTERIOR (HTTPS://GUSTAVOPEIRETTI.COM/PATRON-DE-DISENO-JAVA-FACTORY/)**

ARTÍCULO SIGUIENTE → (HTTPS://GUSTAVOPEIRETTI.COM/PATRON-DE-DISENO-ADAPTER-EN-JAVA/)