TECNOLÓGICO NACIONAL DE MÉXICO
INSTITUTO TECNOLÓGICO DE TIJUANA

SUBDIRECCIÓN ACADÉMICA

DEPARTAMENTO DE SISTEMAS Y COMPUTACIÓN

Tarea
Patrón decorator

Docente
José de Jesús Parra Galaviz

SEMESTRE agosto – diciembre 2021

INGENIERÍA EN TECNOLOGÍAS DE LA INFORMACIÓN Y COMUNICACIONES

Patrones de diseño

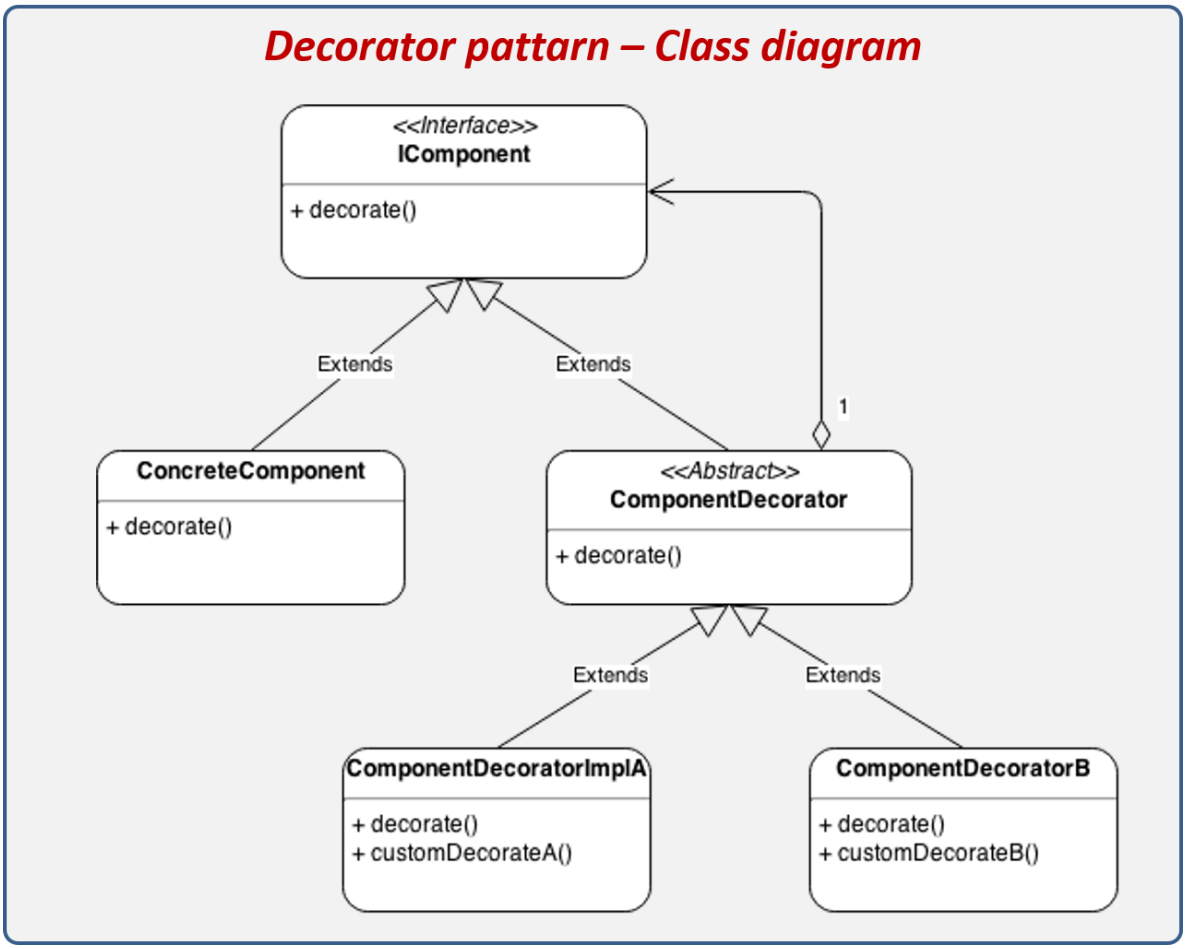
Sergio Alberto Garza Aguilar 15211700

15 de octubre del 2021

Descripción

El patrón decorator está diseñado para solucionar problemas donde la jerarquía con subclasificación no puede ser aplicada, o se requiere de un gran impacto en todas las clases de la jerarquía con el fin de poder lograr el comportamiento esperado. Decorator permite al usuario añadir nuevas funcionalidades a un objeto existente sin alterar su estructura, mediante la adición de nuevas clases que envuelven a la anterior dándole funcionamiento extra.

Diagrama

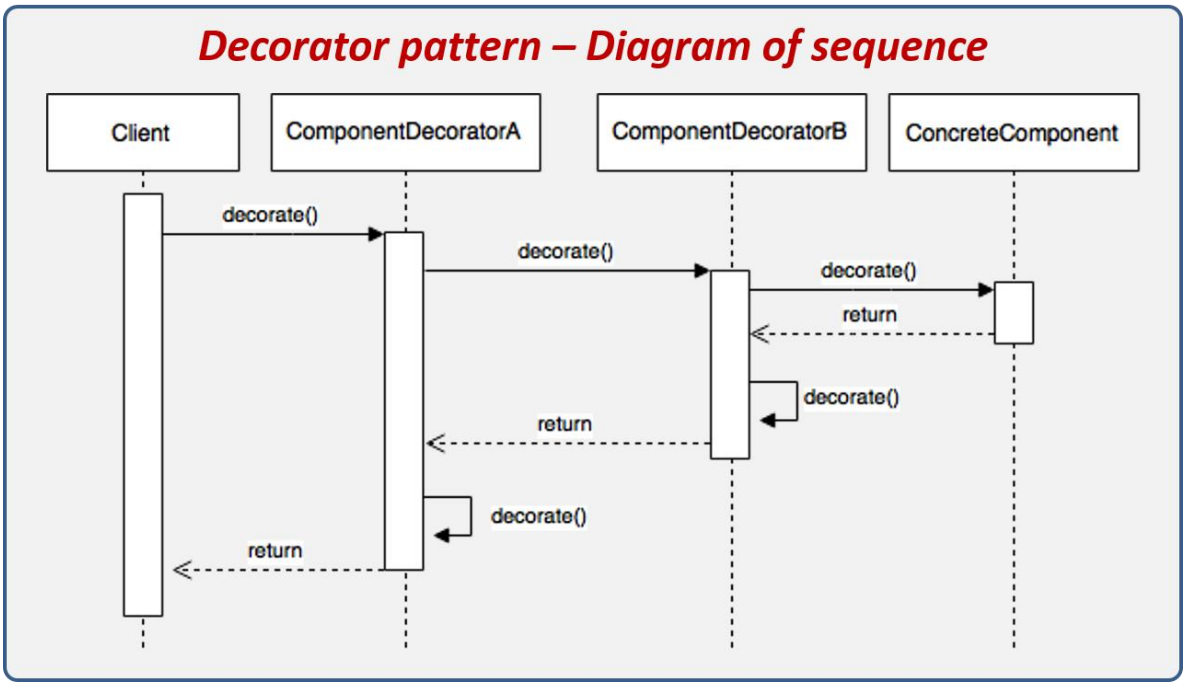


Componentes

- **IComponent**: Interfaz que define la estructura minina del componente o componentes que pueden ser decorados.
- **ConcreteComponent**: Implementación de IComponent y define un objeto concreto que puede ser decorado.

- **ComponentDecorator**: Por lo general es una clase abstracta que define la estructura mínima de un Decorador, el cual mínimamente deben de heredar de IComponent y contener alguna subclase de IComponent al cual decorarán.
- **ComponentDecoratorImpl**: Representan todos los decoradores concretos que heredan de ComponentDecorator.

Diagrama de secuencia



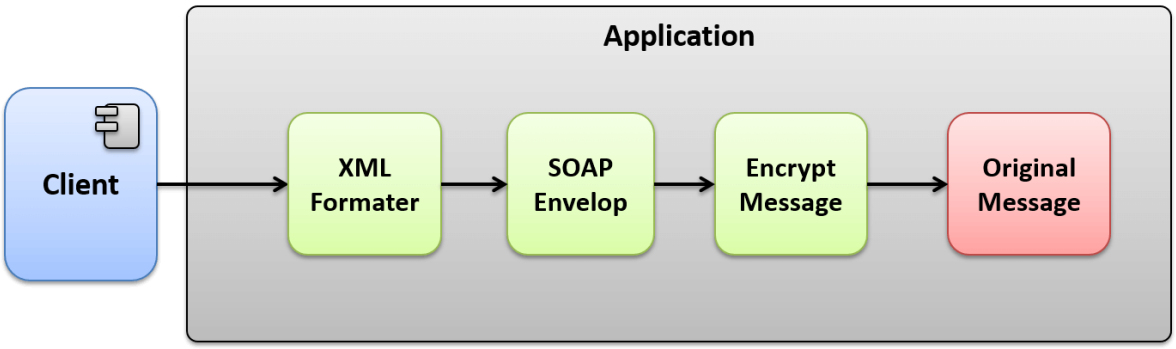
- El Cliente realiza una operación sobre el DecoratorA.
- El DecoratorA realiza la misma operación sobre DecoradorB.
- El decoradorB realiza una acción sobre ConcreteComponente.
- El DecoradorB ejecuta una operación de decoración.
- El DecoradorA ejecuta una operación de decoración.
- El Cliente recibe como resultado un objeto decorado por todos los Decoradores, los cuales encapsularon el Component en varias capas.

Aplicación

Utiliza el patrón Decorator cuando necesites asignar funcionalidades adicionales a objetos durante el tiempo de ejecución sin descomponer el código que utiliza esos objetos.
El patrón Decorator te permite estructurar tu lógica de negocio en capas, crear un decorador para cada capa y componer objetos con varias combinaciones de esta lógica, durante el tiempo de ejecución. El código cliente puede tratar a todos estos objetos de la misma forma, ya que todos siguen una interfaz común.

Utiliza el patrón cuando resulte extraño o no sea posible extender el comportamiento de un objeto utilizando la herencia.
Muchos lenguajes de programación cuentan con la palabra clave *final* que puede utilizarse para evitar que una clase siga extendiéndose. Para una clase final, la única forma de reutilizar el comportamiento existente será envolver la clase con tu propio wrapper, utilizando el patrón Decorator.

Mediante la implementación del patrón de diseño Decorator crearemos una aplicación que nos permite procesar un mensaje en capas, donde cada capa se encargará de procesar un mensaje a diferente nivel. primero convertiremos un Objeto en XML, seguido, lo envolveremos en un mensaje SOAP para después encriptar el mensaje, finalmente obtendremos un mensaje SOAP totalmente encriptado, el cual podrá ser enviado de forma segura a un destinatario. Cada capa de procesamiento será implementada con un decorador, y cada decorador podrá cambiar de posición para obtener un resultado diferente, de la misma manera, podrá ser agregados nuevos decoradores en medio de cualquier paso.



Ventajas

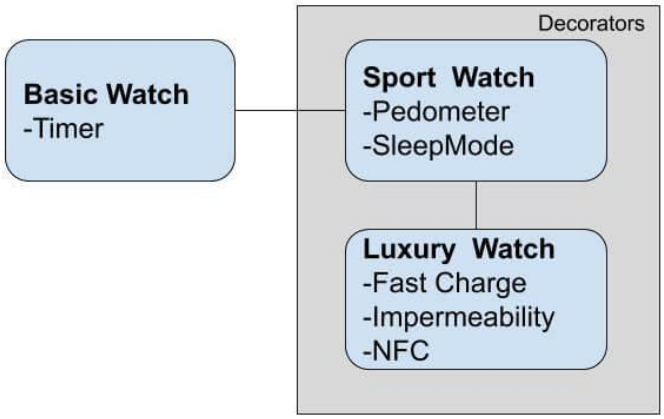
- Puedes extender el comportamiento de un objeto sin crear una nueva subclase.
- Puedes añadir o eliminar responsabilidades de un objeto durante el tiempo de ejecución.
- Puedes combinar varios comportamientos envolviendo un objeto con varios decoradores.
- Principio de responsabilidad única. Puedes dividir una clase monolítica que implementa muchas variantes posibles de comportamiento, en varias clases más pequeñas.

Desventajas

- Resulta difícil eliminar un wrapper específico de la pila de wrappers.
- Es difícil implementar un decorador de tal forma que su comportamiento no dependa del orden en la pila de decoradores.
- El código de configuración inicial de las capas puede tener un aspecto desagradable.

Ejemplo de aplicación

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 los decoradores a partir del decorador principal añadirán las funcionalidades particulares.

```java
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 ");
    }
}
```

```java
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.

```java
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();
    }
}
```

Salida

```
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):  Fas
tCharge  Impermeability  NFC
```

Fuentes

- Oscar Blancarte (08 marzo del 2021), introducción a los patrones de diseño un enfoque practico, Decorator pattern (https://reactiveprogramming.io/blog/es/patrones-de-diseno/decorator)

- Alexander Shvets (2014-2021), Refrescate en los patrones de diseño, Decorator (https://refactoring.guru/es/design-patterns/decorator)

- Gustavo Peiretti (2021), Patron Decorator de diseño en java (https://gustavopeiretti.com/patron-de-diseno-decorator-en-java/)