Taller 5

Ernesto Pérez – 202112530

Germán Moreno – 202116701

Sofia Velásquez – 202113334

Este documento debe combinar textos y explicaciones con diagramas basados en UML. En la mayoría de los casos, esperamos que los diagramas estén acompañados con explicaciones y comentarios que permitan entender claramente lo que ustedes quieren comunicar.

Patrón: **Decorator**

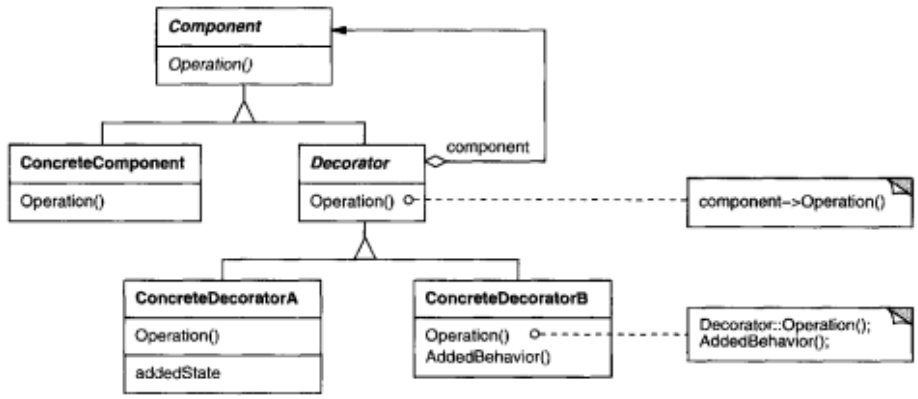
Descripción del patrón:

El patrón Decorator hace parte de los patrones estructurales por lo que ya nos da una idea de que será orientado a la composición de clases y objetos con el fin de construir estructuras más grandes.

Conocido también como 'Wrapper' el este patrón se enfoca en añadir responsabilidades a un objeto de manera dinámica o proveer una alternativa a la herencia para extender la funcionalidad de un programa. (Gang-of-Four. 1995). La razón por la que se usa este patrón es para evitar la herencia puesto que si se requieren nuevas caracterices se violaría el principio *Open Closed* y es poco flexible. Ahora bien, la manera en que este patrón soluciona este problema es mediante encerrando componentes en otros objetos que añadan la característica/ comportamiento.

Concretamente, la aplicación del patrón decorator está en poder añadir responsabilidades a objetos sin afectar otros objetos, para retirar responsabilidades de estos mismos y finalmente cuando la herencia resulta una solución impráctica ya sea que se ya existen muchas subclases u otra razón.

La **estructura** del patrón como establecida en el libro del Gang-of-Four (1995) es la siguiente:



Y sus **componentes** son los siguientes:

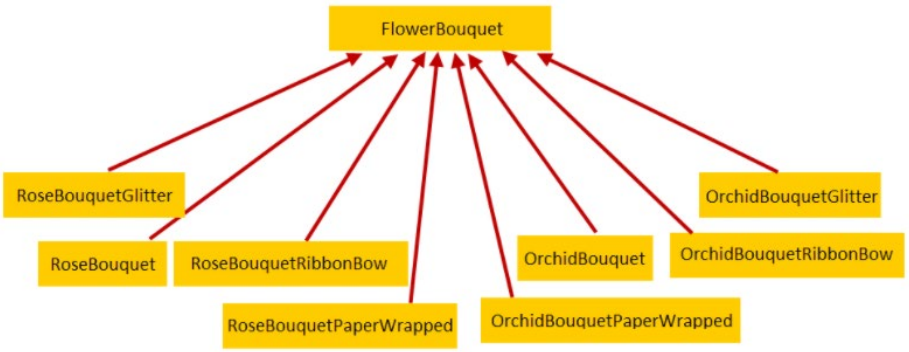
- Component (VisualComponent)
  - define la interfaz para los objetos a los que se pueden agregar responsabilidades dinámicamente.
- ConcreteComponent (TextView)
  - define un objeto al que se pueden adjuntar responsabilidades adicionales.
- Decorator

    - mantiene una referencia a un objeto Componente y define una interfaz que se ajusta a la interfaz del componente.
- ConcreteDecorator (BorderDecorator, ScrollDecorator)
    - Añade responsabilidad a un componente

En cuanto a **colaboraciones** tenemos que el decorador ('Decorator') reenvía las solicitudes a su objeto Componente. Adicionalmente, puede realizar operaciones adicionales antes y después de enviar la solicitud.

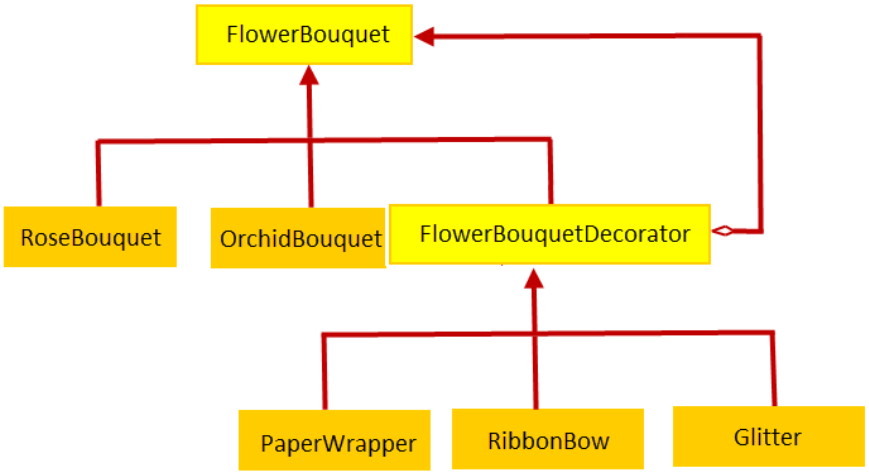
Faltan Ventajas/desventajas + implementacion [Pagina 198 del pdf]

Como ejemplo ilustrativo podemos pensar en el siguiente caso



Aquí tenemos ramos de flores con diferentes características (Tipo de flor, presentación, empaquetado) pero que al ser tantas 'sobrecarga' la clase padre.

Como solución, implementamos el patrón Decorator para decorar dinámicamente nuestro ramo de flores según quiera el usuario con cualquier cantidad de características al momento de la ejecución. De esta manera nuestro diseño queda así:



Y obtenemos unas nuevas relaciones:

- Componente (FlowerBouquet): Es una clase base abstracta que se puede decorar con responsabilidades dinámicamente.

- ConcreteComponent(RoseBouquet y OrchidBouquet): son clases concretas que amplían Component para representar objetos a los que se pueden adjuntar responsabilidades adicionales.
- Decorator (FlowerBouquetDecorator): es una clase abstracta que amplía Component y actúa como clase base para las clases de decorador concretas.
- ConcreteDecorator (PapperWrapper, RibbonBow y Glitter): Son clases concretas que extienden Decorator para decorar Componentes con responsabilidades.

**El ejemplo completo y más información de este se pueden encontrar en:*

https://springframework.guru/gang-of-four-design-patterns/decorator-pattern/

Proyecto seleccionado: https://github.com/mervebasak/Decorator-Pattern-Example/tree/master/src/main/java

Descripción del proyecto:

"Waffle Shop is a café that sells waffles. When customers arrive at this cafe, they design the waffle they want according to the kinds of waffles, their ingredients, and their prices. The waitress delivers these orders to the cook from the customer. The cook is responsible for making the waffle. We will prepare an order system in which we will check the company's waffle orders. We must act with some problems in creating this system. For example, there is a waffle order from the customer. The customer wanted for a waffle with strawberry and white chocolate sauce. We need to expand the functionality of the class in a transparent and dynamic way. Here, we will pay attention to this." Başak, (2019).

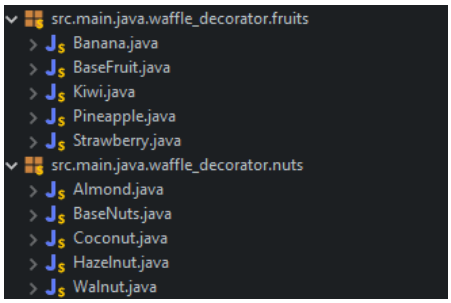
Uso y estructura del patrón dentro del proyecto

Dentro del proyecto, es posible observar que se quiere tener varias opciones para agregar a los wafles. Debido a que hay varias combinaciones posibles, una de las formas más eficientes para agregarlas es el patrón decorator. Dentro del código lo podemos observar dentro de las clases BaseWaffle, BaseFruit, BaseNuts, BaseDecorator y BaseDecorator.

Estas clases son creadas con el propósito de añadir nuevos sabores y opciones a los wafles sin mucho problema, ya que con solo crear una nueva clase que herede una de las mencionadas anteriormente será posible agregar una opción al menú. Por ello es amigable con el principio open-closed.

```
package waffle_decorator.chocolate;

import waffle_decorator.BaseDecorator;
import waffle_dough.BaseWaffle;

public abstract class BaseChocolate extends BaseDecorator {

    public BaseWaffle baseWaffle;

    public BaseChocolate(BaseWaffle waffleToDecorate){
        this.baseWaffle = waffleToDecorate;
    }

    @Override
    public String getDescription() {
        return super.getDescription();
    }
```

En la sección mostrada de código, es posible ver que BaseChocolate hereda sus propiedades de BaseDecorator, al igual que BaseChocolate existen múltiples clases más, que de la misma forma heredan de BaseDecorator, o bien de clases que requieren comportamientos similares como BaseNuts y BaseFruit. Cada una de estas Bases a su vez contiene los sabores disponibles, como lo pueden ser Banana o Kiwi en el caso de las Frutas. Si es necesario, solo hay que crear una nueva clase que extienda la clase principal (BaseFruit) para agregar una nueva.



Análisis

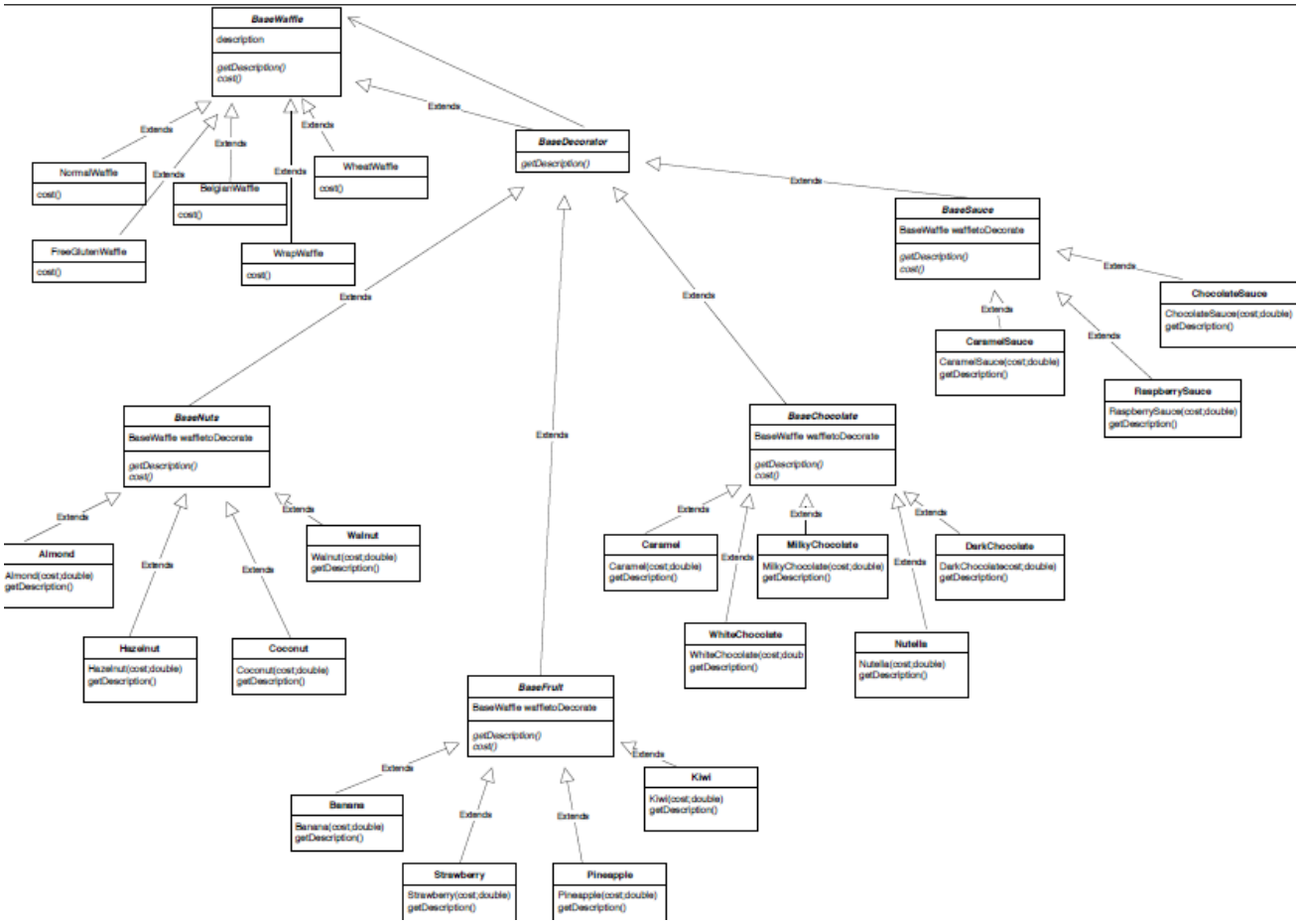
- Información y estructura del fragmento del proyecto donde aparece el patrón. No se limiten únicamente a los elementos que hacen parte del patrón: para que tenga sentido su uso, probablemente van a tener que incluir elementos cercanos que sirvan para contextualizarlo.
- Información del patrón aplicado al proyecto: explicar cómo se está utilizando el patrón dentro del proyecto
- ¿Por qué tiene sentido haber utilizado el patrón en ese punto del proyecto? ¿Qué ventajas tiene?
- ¿Qué desventajas tiene haber utilizado el patrón en ese punto del proyecto?

Comparando nuestro proyecto con el ejemplo utilizado para describir el patrón (ramos de flores), vemos que es en ultimas estamos haciendo lo mismo: a un objeto inicial (Flores/waffles) modificándolos y añadiéndoles varias características y componentes.

El haber utilizado el patrón decorator nos permite tener nuestro waffles básico (Normal, belgian, wrap, etc) como un componente concreto y añadirles a estos decoradores como chocolate, fruta, salsa

y masa, cada uno con diferentes tipos a su vez. Así podemos modificar el objeto 'BaseWaffle' pero no directamente puesto que está encerrado en los decoradores, esto es casi como si nuestro objeto no sabe que está siendo modificado. Aparte de que podemos modificar nuestro objeto en tiempo de ejecución, con esta implementación resulta muy facil añadir más tipos de objetos que hagan parte de un decorador (como chocolates) o nuevos decoradores (Ej: si quisiéramos agregarle diferentes formas de presentación al waffle)

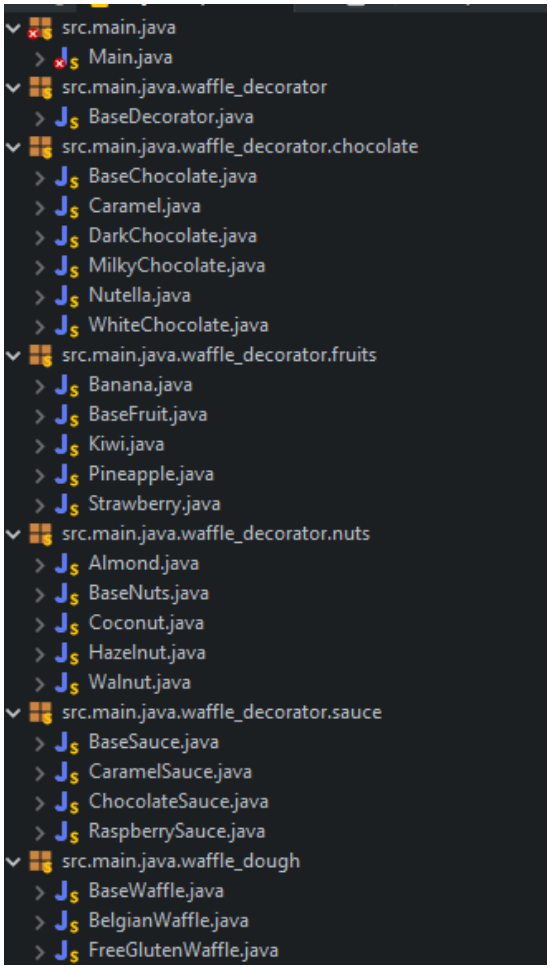
Para ilustrar lo que se dijo anteriormente está el diagrama UML del proyecto a continuación donde es clara la relación en entre nuestro objeto y los diferentes decoradores que tiene:



**El UML completo está en el repositorio del proyecto donde es más nítido y en formato pdf.*

Aquí se ve como nuestro waffle es padre de los diferentes tipos de waffles y una vez esto es definido esta un decorador base que extiende a decoradores más pequeños ya con lo que se le quiera agregar.

Mas allá, vemos también que en la estructura del proyecto se ve el uso del patrón decorator en cómo se organizan las clases y los paquetes en los que están.



Se ve claramente que hace parte de cada decorador y que el decorador base es su paquete independiente.

Entrando en ventajas y desventajas

Ya se mencionó que sus ventajas corresponden a la practicidad de tener un decorador que nos modifique nuestro objeto en tiempo de ejecución y que es muy flexible a la hora de tener más características (o menos) en nuestro objeto pues es simplemente hacer unas modificaciones pequeñas en el decorador base que no afectan a nuestro objeto, mucho más flexible que dejáramos que nuestra clase de waffles heredara todas las modificaciones, por ejemplo-

Por otro lado, una posible desventaja es que resultamos con muchos objetos pequeños pero muy parecidos que son diferentes unos a otros solamente en la manera en que están conectados o a que decorador pertenecen, si bien esto lo vuelve altamente personalizable es algo difícil de entender y mucho más difíciles de debugear cuando se generen problemas. Adicionalmente, toca tener cuidado en que nuestro objeto 'decorado' (es decir con las modificaciones del decorator) no es idéntico al objeto original por lo que puede ser difícil de manejar.

¿De qué otra manera pudo ser resuelta la problemática resuelta por el patrón?

Para solucionar esto, podrían ser creadas múltiples clases, cada una que se encargara de agregar un nuevo sabor, sin necesidad de heredar de la clase principal. De manera tal que para agregar un nuevo sabor se creara una nueva clase específica al mismo, y luego fuera importada dentro de la clase que se necesita, en este caso la clase waffle, cada que se creara

un nuevo sabor, o una opción de cubierta se realizaría este proceso. El resultado que se obtiene es el mismo, la única desventaja de este método es que a pesar de tener bajo acoplamiento también terminamos con una cohesión muy baja y agregar nuevas características a todas las clases de sabores termina en una situación más difícil de manejar.

Bibliografía

1. Başak, Merve (2019) Decorator-Pattern-Example [https://github.com/mervebasak/Decorator-Pattern-Example#decorator-pattern-example]
2. https://springframework.guru/gang-of-four-design-patterns/decorator-pattern/
3. Gamma, E., Helm, R., Johnson, R., Vlissides, J., & Patterns, D. (1995). Elements of Reusable Object-Oriented Software. *Design Patterns. massachusetts: Addison-Wesley Publishing Company*. http://www.javier8a.com/itc/bd1/articulo.pdf