

1. Información general de proyecto: Waffle Shop

1.1

URL: <https://github.com/mervebasak/Decorator-Pattern-Example.git>

1.2

Contexto:

Waffle Shop es un café que vende waffles. Cuando los clientes llegan a este café, diseñan el waffle que desean según los tipos de waffles, sus ingredientes y sus precios. La camarera entrega estos pedidos al cocinero del cliente. El cocinero es responsable de hacer el waffle. Prepararemos un sistema de pedidos en el que revisaremos los pedidos de waffles de la compañía. Tenemos que actuar con algunos problemas al crear este sistema. Por ejemplo, hay un pedido de waffle de un cliente. El cliente quería un waffle con salsa de fresa y chocolate blanco. Necesitamos expandir la funcionalidad de la clase de una manera transparente y dinámica

2. Patrón:Decorator

2.1 Contexto Explicación de los elementos:

El patrón de diseño Decorator se aplica para resolver problemas existentes en el proyecto. Este patrón permite cambiar dinámicamente el comportamiento de un objeto en tiempo de ejecución envolviéndolo en un objeto de una clase Decoradora. En el mundo real, imagina que diriges un taller de servicios para automóviles que ofrece múltiples servicios. ¿Cómo calculas la factura a cobrar? Eliges un servicio y dinámicamente le vas añadiendo los precios de los servicios proporcionados hasta obtener el costo final. Aquí, cada tipo de servicio es un Decorator.

Información Importante:

Flexibilidad: El patrón Decorator ofrece una gran flexibilidad al permitir modificar el comportamiento de un objeto sin alterar su código fuente o cambiar el comportamiento de objetos individuales sin afectar a otros de la misma clase.

Evita Clases Complejas: En lugar de crear clases que soporten una combinación de comportamientos, puedes combinar funcionalidades mediante la decoración, evitando así la necesidad de clases con muchas responsabilidades o capacidades.

Composición sobre Herencia: Este patrón favorece la composición sobre la herencia, permitiendo un diseño más fluido y evitando problemas relacionados con jerarquías de herencia profundas y rígidas.

Responsabilidad Única: Cada Decorator se centra en una funcionalidad específica, manteniendo la cohesión y respetando el principio de responsabilidad única.

Desacoplamiento: Los Decoradores pueden ser añadidos o eliminados en tiempo de ejecución, lo que proporciona una mayor separación entre la funcionalidad básica del objeto y sus extensiones o modificaciones.

Transparencia: Desde el punto de vista del cliente, el uso de Decoradores debería ser transparente; el cliente no necesita saber si está trabajando con un objeto decorado o no.

Este patrón es especialmente útil en situaciones donde se necesita añadir responsabilidades a objetos de manera dinámica y se quiere mantener el código flexible y reutilizable. Sin embargo, puede aumentar la complejidad y llevar a una gran cantidad de pequeñas clases, lo cual puede ser difícil de entender y gestionar.

2.2

Motivación:

Este patrón es ideal porque me permite ofrecer una amplia variedad de combinaciones sin necesidad de crear una clase diferente para cada posible combinación de waffle. Además, a medida que amplíé mi menú con nuevos sabores y toppings, el patrón Decorator facilita la adición de nuevos Decorators sin alterar las clases existentes, lo que hace que mi sistema sea escalable y fácil de mantener. En resumen, el patrón Decorator da la flexibilidad y la extensibilidad que se necesita para satisfacer las preferencias individuales, manteniendo al mismo tiempo un código limpio y organizado.

3. Aplicación del patrón

3.1

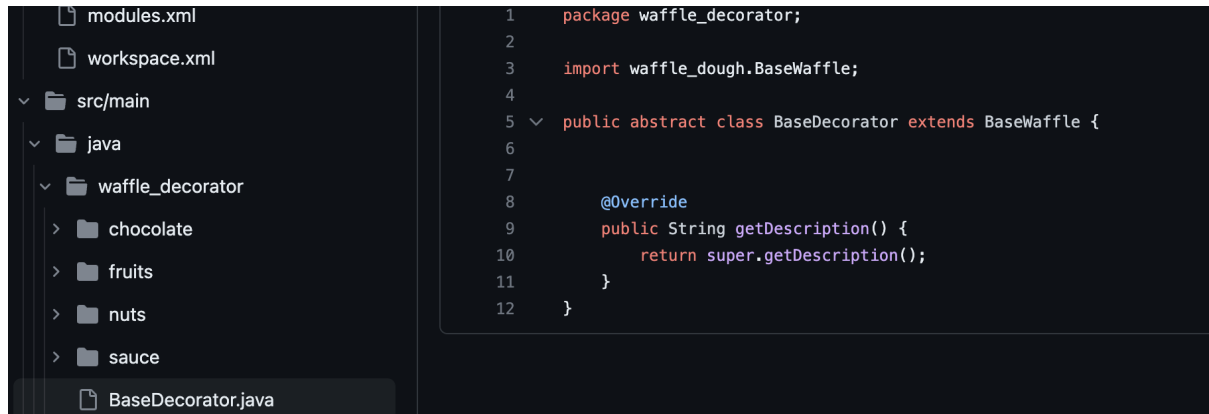
Como se utiliza en el proyecto:

En la tienda de waffles, se aplica el patrón Decorator para habilitar una personalización dinámica y única para cada cliente. Los visitantes de la tienda seleccionan un waffle base de las opciones disponibles, a lo que luego pueden añadir una variedad de frutas, salsas y otros ingredientes según su preferencia. Se establecen clases base de waffles como `ComponenteConcreto`, mientras que los elementos adicionales como chocolate, frutas y salsas funcionan como `DecoradoresConcretos`. Por ejemplo, en un pedido típico, el cliente comienza eligiendo la masa del waffle. Después, puede agregarle chocolate Nutella y seleccionar frutas como fresas, plátanos o piña. Para finalizar, tiene la opción de añadir salsa de chocolate blanco y trozos de nuez o almendra. Este enfoque no solo satisface las preferencias individuales de los clientes sino que también mantiene la estructura del menú organizada y flexible para futuras expansiones o modificaciones.

4. Aparición del patrón

BaseDecorator.java

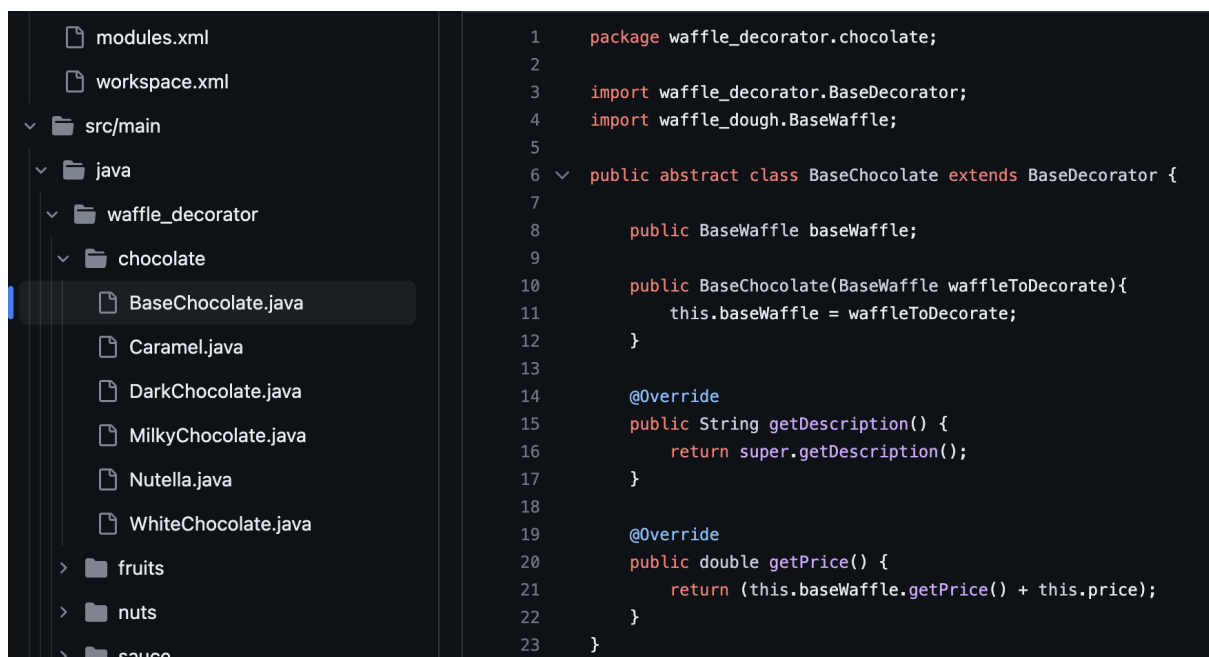
Esta clase es una abstracción y extiende BaseWaffle, es una clase que define la funcionalidad básica de un waffle. BaseDecorator actúa como la superclase para todos los decoradores concretos (como los sabores de chocolate, frutas, etc.). Sobrescribe el método getDescription(), que está llamando al método de la superclase BaseWaffle.



```
1 package waffle_decorator;
2
3 import waffle_dough.BaseWaffle;
4
5 public abstract class BaseDecorator extends BaseWaffle {
6
7
8     @Override
9     public String getDescription() {
10         return super.getDescription();
11     }
12 }
```

BaseChocolate.java

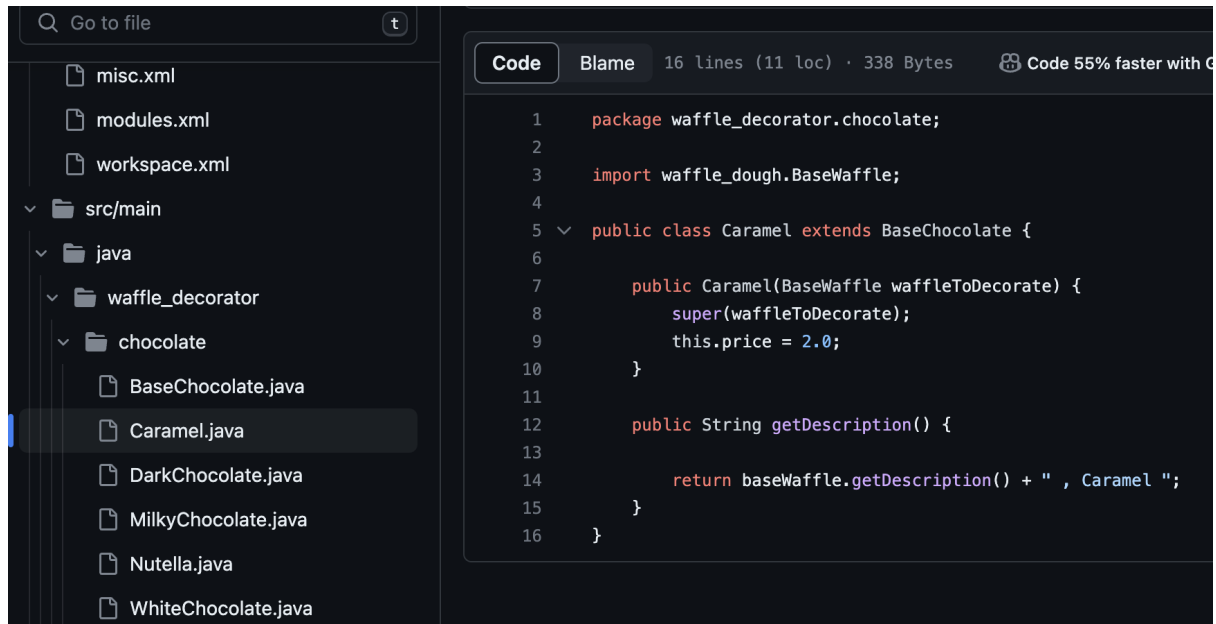
BaseChocolate es una clase abstracta que extiende BaseDecorator, y es la superclase para tipos específicos de decoradores de chocolate. Contiene una referencia baseWaffle al objeto BaseWaffle que está decorando y define un constructor que acepta un BaseWaffle como parámetro. Además, implementa getDescription() y getPrice() que son métodos que son usados para obtener la descripción y el precio del waffle con el decorador de chocolate aplicado.



```
1 package waffle_decorator.chocolate;
2
3 import waffle_decorator.BaseDecorator;
4 import waffle_dough.BaseWaffle;
5
6 public abstract class BaseChocolate extends BaseDecorator {
7
8     public BaseWaffle baseWaffle;
9
10     public BaseChocolate(BaseWaffle waffleToDecorate){
11         this.baseWaffle = waffleToDecorate;
12     }
13
14     @Override
15     public String getDescription() {
16         return super.getDescription();
17     }
18
19     @Override
20     public double getPrice() {
21         return (this.baseWaffle.getPrice() + this.price);
22     }
23 }
```

Caramel.java

Esta es una clase concreta que extiende BaseChocolate, representando un tipo específico de decorador de chocolate (en este caso, caramelo). Define un constructor que toma un BaseWaffle y pasa este waffle al constructor de la superclase BaseChocolate. También establece un precio para el decorador de caramelo. Sobrescribe getDescription() para devolver la descripción del waffle decorado más la cadena " , Caramel", lo que indica que el caramelo ha sido añadido al waffle.



```
1 package waffle_decorator.chocolate;
2
3 import waffle_dough.BaseWaffle;
4
5 public class Caramel extends BaseChocolate {
6
7     public Caramel(BaseWaffle waffleToDecorate) {
8         super(waffleToDecorate);
9         this.price = 2.0;
10    }
11
12    public String getDescription() {
13
14        return baseWaffle.getDescription() + " , Caramel ";
15    }
16 }
```

5. Ventajas

- **Flexibilidad:** Permite añadir dinámicamente nuevas funcionalidades (decoraciones en este caso) a los objetos en tiempo de ejecución. Esto es particularmente útil en la tienda de waffles, donde cada cliente puede tener un pedido único.
- **Reutilización:** Los decoradores pueden aplicarse en diferentes combinaciones sobre cualquier BaseWaffle, lo que promueve la reutilización del código. Por ejemplo, la misma clase Caramel se puede utilizar para decorar cualquier tipo de waffle sin necesidad de duplicar código.
- **Extensibilidad:** Se pueden agregar nuevos decoradores sin modificar el código existente. Si la tienda decide introducir un nuevo sabor o topping, simplemente puede crear una nueva clase de decorador sin afectar las clases existentes.
- **Separación de Responsabilidades:** Cada clase tiene una única responsabilidad. Por ejemplo, Caramel solo maneja la lógica relacionada con la adición de caramelo. Esto facilita el mantenimiento y la comprensión del código.
- **Evita Clases con Alta Granularidad:** En lugar de crear una clase para cada combinación posible de waffle y sus decoraciones, el patrón Decorator permite combinar funcionalidades de manera más eficiente.

- **Combinación de Comportamientos:** Mediante el uso de decoradores, es posible combinar múltiples comportamientos (toppings, salsas, etc.) de manera que cada uno contribuya con su parte sin crear dependencias complejas entre ellos.

6. Desventajas

Dificultades en la configuración: Configurar un objeto con múltiples capas de decoradores puede volverse engorroso y propenso a errores,

Problemas de rendimiento: La creación de múltiples objetos decoradores puede afectar el rendimiento si el proceso de decoración se realiza muchas veces o en objetos que son muy utilizados. Esto también puede aumentar el uso de la memoria.