Taller 5 DPOO

Elaborado por: Hernán Ramírez - 202124034

- **Waffle Shop**

El proyecto que voy a analizar a lo largo de este documento se llama “Waffle Shop”, el cual representa una tienda que vende wafles que pueden ser personalizados por el cliente. Al momento de ingresar en la tienda, el cliente diseña el wafle que desea comprar de acuerdo con la base, los ingredientes, las salsas, etc. El mesero luego lleva la orden al chef, quien lo prepara. Además, la aplicación implementa un sistema que permite verificar los pedidos de la tienda, donde un pedido puede tener la siguiente forma: wafle con fresas y salsa de chocolate (salvo que se indique lo contrario, se trata de un wafle básico).

En concreto se identificaron 4 requerimientos funcionales para este proyecto:

1) El cliente debe poder diseñar un wafle (forma e ingredientes).
2) El mesero toma la orden del cliente y la entrega al chef.
3) El mesero almacena la orden para que esta pueda ser consultada.
4) El chef recibe la orden del cliente y prepara un wafle acorde a esta.

Respecto al principal desafío que plantea la autora del proyecto esta poder actualizar de manera dinámica y transparente las funcionalidades de las clases. Esto hace referencia a poder actualizar los diferentes ingredientes adicionales que ofrece la tienda y que esto se vea reflejado en las opciones de personalización del wafle.

- **Patrón de diseño: Decorator**

El patrón de diseño sobre el cual se enfoca el proyecto es el decorator, por lo cual , antes de detallar como este fue implementado, es necesario comprender en que se basa este patrón y que clase de problemas soluciona.

Decorator, tambien conocido como Wrapper, es un patrón de diseño que busca adjuntar responsabilidades adicionales a un objeto de manera dinámica. Esto quiere decir que es una alternativa flexible a crear subclases con el fin de extender funcionalidades. La manera en que lo anterior se logra es a través de poner objetos llamados “decoradores” sobre objetos base sin alterar el comportamiento base de estos últimos. Adicionalmente, véase que esto permite modificar el comportamiento de instancias directamente sin modificar a toda la clase.

Un claro ejemplo de esto en la vida real es el uso de ropa y accesorios. Concretamente, la ropa que alguien utilice puede depender del entorno en el que se encuentre; por ejemplo, si hace calor puede remover capas de prendas o utilizar ropa de mangas cortas o bermudas; igualmente alguien podría hacer lo inverso en caso de estar en un entro frio, optando por ropa más abrigada y larga; o por ejemplo en el caso de lluvia se podría añadir un impermeable sobre la ropa que ya se lleve o se podría añadir un paraguas. En todos los ejemplos descritos anteriormente el tipo de ropa extiende los comportamientos básicos sin ser una parte intrínseca de la persona.

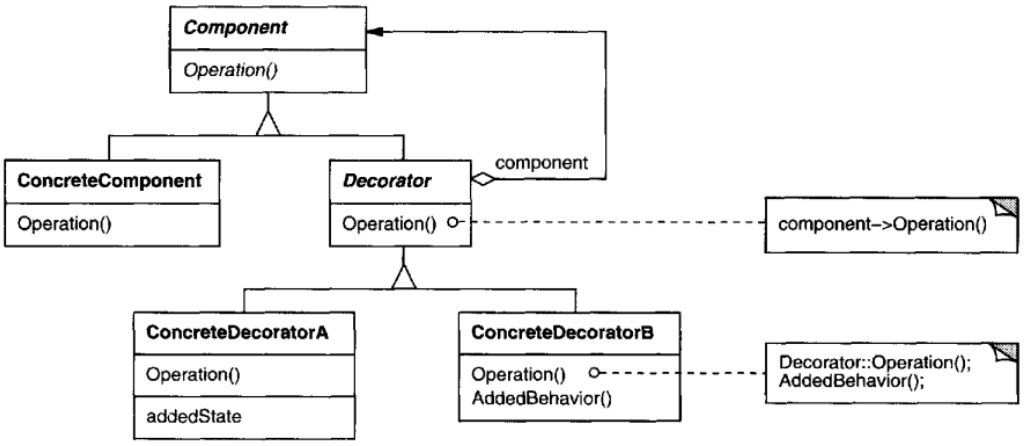

Ilustración 1Diagrama UML para el patrón Decorator [1]

Respecto a la implementación de este patrón de diseño esta cuenta con los siguientes componentes:

- Componente: Define la interfaz común para todos los objetos cuyas responsabilidades pueden ser actualizadas dinámicamente.
- Componente Concreto: Implementa la interfaz del componente y define el objeto al que se le pueden agregar responsabilidades.
- Decorador: También implementa la interfaz del componente, pero contiene una referencia a un componente y agrega responsabilidades adicionales de manera dinámica.
- Componente Decorado: Añade responsabilidades al componente

Por último, respecto a los beneficios que ofrece este patrón es que permite actualizar objetos durante la ejecución, pues simplemente sería necesario "conectar" o "desconectar" al decorador en cuestión; en contraposición con la herencia, la cual solo podría ser modificada fuera de la ejecución. Además, permite evitar que las clases mas altas de la jerarquía queden cargadas de muchas funcionalidades que realmente son de interés solo para algunas de sus subclases que pueden estar muy abajo en la jerarquía.

- **Decorator en Waffle Shop**

Una vez se entendió el patrón del decorator es claro porque este fue utilizado en este proyecto, pues este patrón sería muy para implementar adiciones a un producto, que en este caso serían los wafles. Esto se debe a que, en ultimas, el cliente esta comprando un wafle independientemente de todos los ingredientes extra que desee añadir; dicho de otra forma, un pedido puede incluir únicamente el wafle básico o un wafle con todas las adiciones posibles, pero en ambos casos el wafle no pierde sus propiedades básicas, simplemente se están añadiendo extras sobre él.

Mas concretamente, en el proyecto el proyecto seleccionado incluye un diagrama de clases donde se muestra la implementación del patrón:

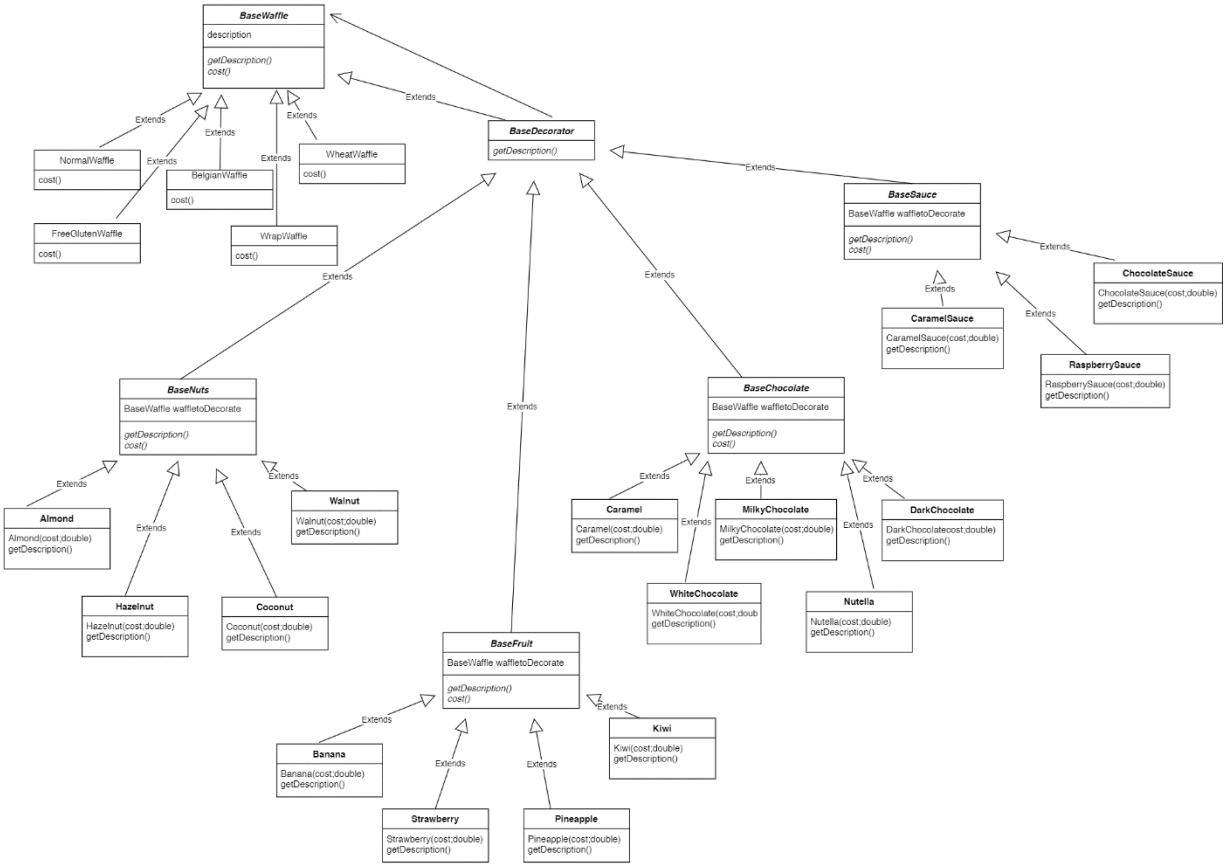

Ilustración 2Diagrama UML del proyecto Waffle Shop [2]

En este diagrama es claro que existen múltiples decoradores para un wafle, pues incluso estos están organizados por diferentes categorías. Al analizar el código del proyecto, se puede notar que la clase BaseWaffle cumple el papel de Componente, aunque no sea del tipo interfaz sino que es una clase abstracta. A su vez, esto indica que las clases NormalWaffle, FreeGlutenWaffle, BelgianWaffle, WrapWaffle y WheatWaffle son los componentes concretos sobre los cuales se van a poner los decoradores. Así mismo, se tiene la clase BaseDecorator la cual se conecta a BaseNuts, BaseFruit, BaseChocolate y BaseSauce, las cuales actúan como Decorators encargados de referenciar al waffle para que se puedan poner sobre él las adiciones, que en este caso corresponden a las clases conectadas en cada caso, como Banana, Strawberry, Pineapple y Kiwi en el caso de BaseFruit; donde estas últimas son los decoradores concretos en este caso.

- **Ventajas y Desventajas de usar Decorator en Waffle Shop**

A simple vista, el uso de Decorator en este proyecto parece una opción viable y, de hecho, uno de los uso mas comunes para este tipo de problemas. A continuación se discuten algunas de las ventajas y desventajas de haber utilizado este patrón en Waffle shop con el fin de determinar si era la opción más acertada.

En primer lugar, una de las principales ventajas que ofrece este patrón es que permite que el comportamiento de la clase que se busca extender pueda ser alterado durante la ejecución del programa pues, como se mencionó antes, simplemente sería necesario conectar o desconectar los decoradores. Esto es muy útil en este caso pues permitiría que el cliente pudiese modificar su orden con facilidad y sin sobre complicar el código a través de relaciones de herencia. En segundo lugar, este patrón permite que la tienda pueda extender su catalogo de ingredientes extra, pues simplemente sería necesario conectar los nuevos ingredientes a la interfaz correspondiente (frutas, nueces, salasas, etc.) o en su defecto implementar una nueva interfaz que se conecte al BaseDecorator (por ejemplo BaseDulces que incluya gomitas, m&m`s, masmelos, etc.). Por último, la principal desventaja sería que la cantidad de clases podría escalar considerablemente ya que existiría al menos una por cada ingrediente extra que se quisiera añadir, lo cual podría dificultar el mantenimiento del código; sin embargo, dado el tamaño del proyecto estas clases no deberían ser difíciles de manejar ni deberían generar confusión al momento de implementarlas pues el diseño actual es bastante coherente por si mismo.

- **Otras alternativas que sustituyen Decorator**

Existen diferentes alternativas que podrían sustituir al patrón de Decorator sin necesidad de recurrir al uso de herencia. Un patrón de diseño que lograría esto sería Chain of Responsability, el cual en lugar de agregar responsabilidades dinámicamente a un objeto, el patrón Chain of Responsibility organiza objetos en una cadena y permite que cada objeto en la cadena tenga la oportunidad de manejar la solicitud.

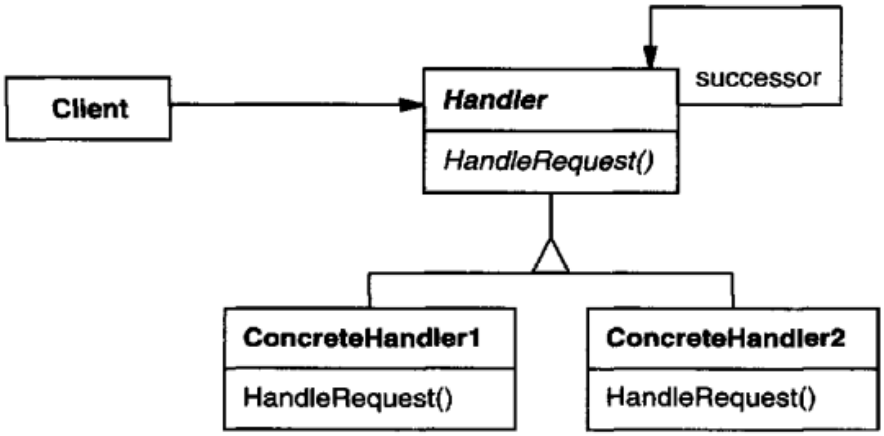

Ilustración 3Diagrama UML para el patrón Chain of Responsibility [1]

De esta forma, podría implementarse una cadena de responsabilidades que permitan manejar la informacion de cada pedido según lo que desee el cliente.

No obstante, consideró que la siguiente opción más viable sería utilizar el patrón de Composite, el cual permite establecer relaciones entre objetos donde un objeto se compone de otros. Es decir, que se podría tener un objeto “Wafle” compuesto por diferentes objetos de distintas clases que serian los ingredientes en cada caso. No obstante, esto produce el inconveniente de que se perdería el concepto de waffle como base primordial del pedido, pues bajo el patrón de Composite

no tiene sentido hablar de un objeto compuesto sin componentes, pero las componentes por si mismas pueden tener sentido. Una posible implementación podría ser la siguiente :

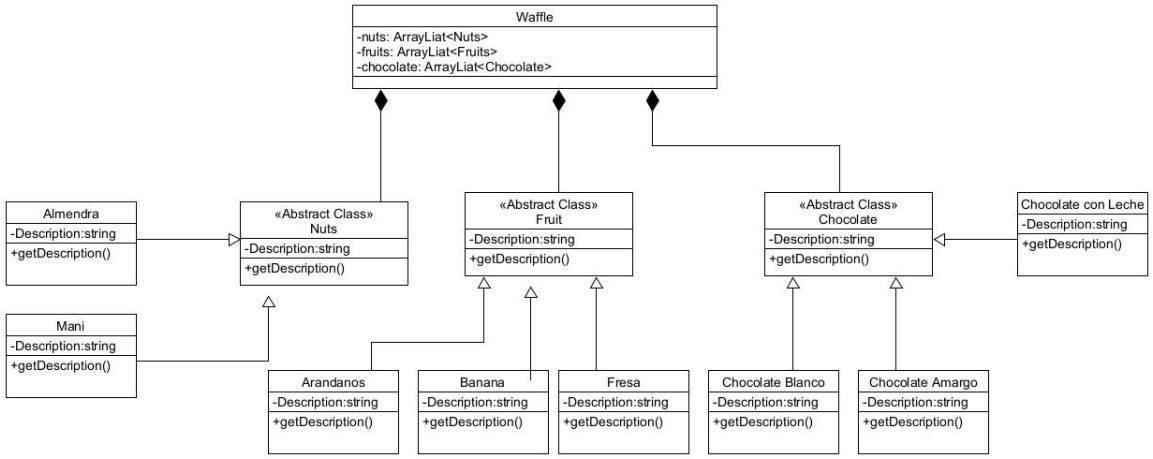

Ilustración 4 Diagrama UML alternativa Composite

No obstante, véase que para manejar el caso en el que el cliente no elige ningún ingrediente extra se debería implementar alguna clase que represente no añadir nada.

- **Referencias:**

[1] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. Consultado en línea https://www.javier8a.com/itc/bd1/articulo.pdf

[2] mervebasak. (2019). Decorator-Pattern-Example.GitHub. https://github.com/mervebasak/Decorator-Pattern-Example.git