Proyecto sobre patrones de diseño

UPOFOOD

### Grupo 15

Javier Merino Jabalera

Carlos Lacave Muñoz

Víctor Ballesteros León

## Concepto principal del proyecto

En este proyecto queremos realizar la gestión de un restaurante de comida rápida a domicilio, donde se sirven varios tipos de comida.

El restaurante tiene un grupo de cocineros que se encargan de realizar los pedidos de los clientes. Además, hay un administrador que puede contratar, despedir o consultar los cocineros que tiene el restaurante, además de poder gestionar también los clientes.

Por otro lado, los clientes pueden tener una lista de pedidos favoritos para una mayor rapidez a la hora de realizar dichos pedidos. Cuando un cliente va a realizar un pedido, este puede hacerlo desde la lista de favoritos o crear uno nuevo. Los clientes también cuentan con una opción para visualizar sus pedidos favoritos y cambiar la contraseña de su usuario si así lo desean.

## Patrones

Para resolver este problema utilizamos una serie de patrones de diseño software que ayudan a tener una solución general, reutilizable y que puede aplicarse a varios problemas de diseño software.

Hemos utilizado 5 patrones distintos: Estrategia, Observador, Singleton, Factoría y Fachada.

### Patrón Estrategia

Permite desarrollar un solo método en una interfaz, la cual se implementaría en otras clases/subclases, y permitirá que el método se vaya modificando en tiempo real dependiendo de la clase en la que se llame, sin tener que implementar un método diferente para cada clase.

Ha sido implementado en nuestro caso para seleccionar el complemento del menú mediante la selección de una estrategia al crear una comida.

Definimos el comportamiento de un método común para todas las comidas durante la ejecución del código.

### Patrón Observador

El patrón Observador permite definir un mecanismo de observación para notificar a varios objetos sobre cualquier evento que le suceda al objeto que están observando.

En el patrón observador de nuestra solución participan los cocineros como observadores, que son notificados cuando se realiza un cambio en una instancia de pedido, en nuestro caso cuando el cliente confirma el pedido.

Por otro lado el pedido actúa como sujeto observado, teniendo una lista de observadores(cocineros) que están a la espera de la confirmación(notificar observadores) y se les notifique el cambio con las comidas a preparar.

### Patrón Singleton

Nos permite tener únicamente una sola instancia de la clase de la clase con la que estamos trabajando. Al tener una clase de tipo Singleton no podremos crear más de una instancia de dicha clase, en su lugar si intentamos crear nuevas instancias devolverá la instancia ya creada anteriormente.

En el caso de UPOFood tenemos dos clases de tipo Singleton, la clase Restaurante y la clase Administrador. Por el motivo explicado anteriormente nuestro programa tendrá una única instancia de Restaurante y Administrador.

### Patrón Factoría

El patrón Factoría nos permite crear diferentes tipos de comida de manera eficiente.

Todos los tipos de comida tienen una interfaz común, en el caso de la pizza, todos los tipos cuentan con un precio, un nombre, una lista de ingredientes, etc. Esto nos permite crear clases separadas para los distintos tipos de pizza. Centralizar las pizzas o cualquier comida en una clase común nos permite añadir más comidas de forma fácil y eficiente, y se podrían encapsular las complejidades de la creación de objetos en un número menor de clases.

### Patrón Fachada

El uso del patrón Fachada nos proporciona una interfaz simple y unificada tanto para los clientes como para el administrador del restaurante. Esta clase se encarga de encapsular la complejidad de las diferentes funcionalidades del sistema.

De esta manera, tanto los clientes como el administrador del restaurante no tendrían que conocer la complejidad interna del sistema y permite realizar las operaciones de una manera más simple. Además, si se realizan cambios internos en el sistema, solo tendrían que realizarse pequeños cambios en la fachada en lugar de tener que modificar el código en todos los lugares que se usase dicha clase.

### \*CORRECCIÓN Y MODIFICACIONES\*

* Se han corregido algunos errores en relación al patrón observador y las clases que implementan la interfaz EstrategiaPreparacion.java.
* Se ha añadido una nueva funcionalidad que modifica el precio de un menú al añadir patatas o refresco, además de añadirlas a la lista de ingredientes en lugar de mostrar un mensaje por pantalla.

Se ha adaptado el diagrama UML y los documentos a estos cambios y correcciones pero es posible que alguna de las imágenes mostradas en alguno de los documentos no corresponda a la versión final del código, pero el concepto general sigue siendo el mismo.