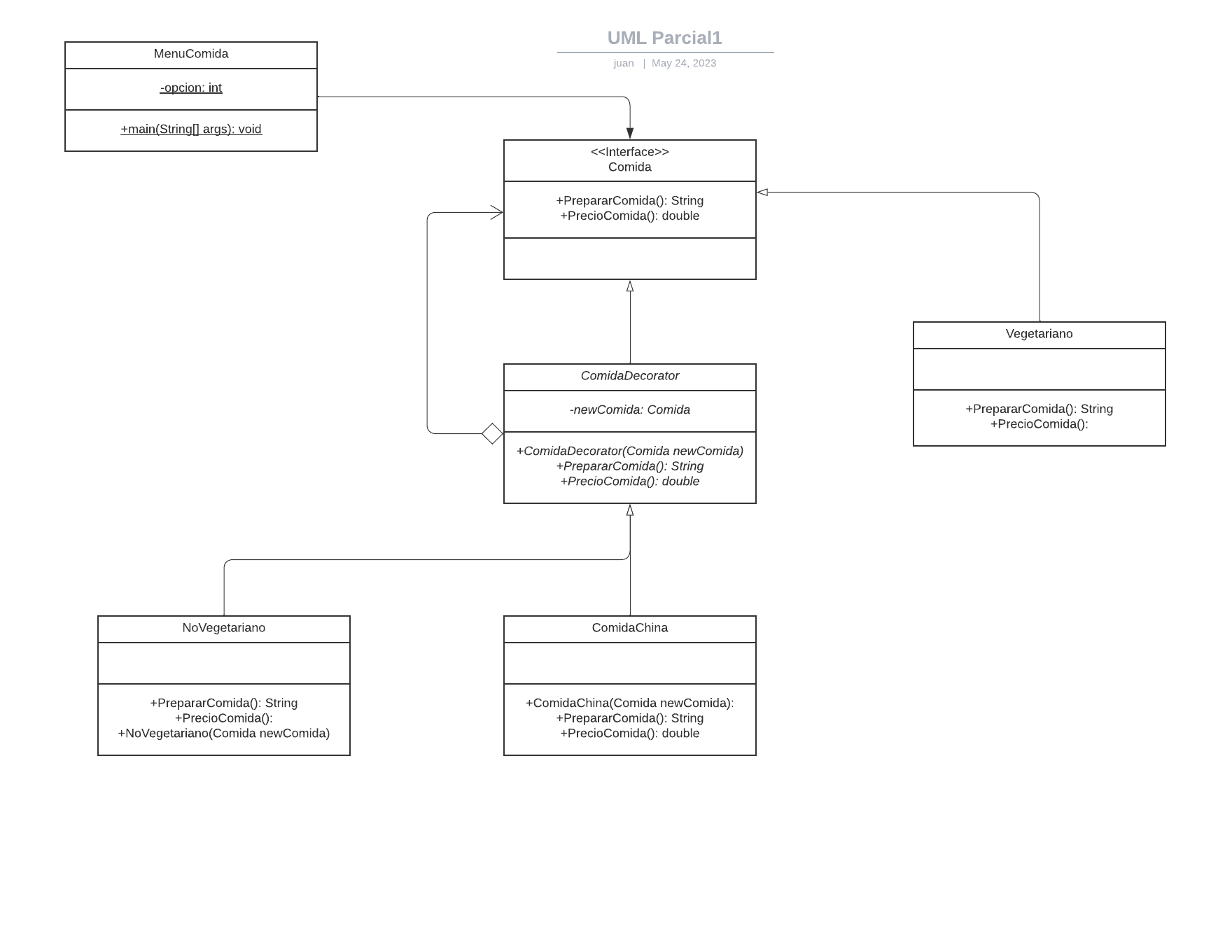
El patrón elegido es el “Decorador”, este patrón se basa en un diseño estructural que te permite añadir funcionalidades a objetos colocando estos objetos dentro de objetos encapsuladores especiales que contienen estas funcionalidades. “Wrapper” (envoltorio, en inglés) es el sobrenombre alternativo del patrón Decorator, que expresa claramente su idea principal. Un wrapper es un objeto que puede vincularse con un objeto objetivo. El wrapper contiene el mismo grupo de métodos que el objetivo y le delega todas las solicitudes que recibe. No obstante, el wrapper puede alterar el resultado haciendo algo antes o después de pasar la solicitud al objetivo. Estos Wrappers se va guardando en una pila de decoradores.

Ahora bien, se comenzará a analizar el patrón por medio de un proyecto investigado en GitHub, el enlace de este se deja a continuación: https://github.com/Derliodev/PatronDecorator.git

En ese orden de ideas, se presenta el diagrama UML del proyecto el cual muestra todas las clases que lo complementan y de qué manera se relacionan para demostrar el patrón decorador.



Este proyecto muestra que el patrón se representa desde la clase ComidaDecorator. Lo que hace esta clase es realizar una herencia a la clase Comida para crear un nuevo objeto, en este caso se implementa para preparar ComidaChina o NoVegetariano y entregar el precio de cada uno.

public abstract class ComidaDecorator implements Comida {

private Comida newComida;

public ComidaDecorator(Comida newComida){

this.newComida = newComida;

}

*@*Override

public String PrepararComida(){

return newComida.PrepararComida();

}

*@*Override

public double PrecioComida(){

return newComida.PrecioComida();

}

}

public class MenuComida {

private static int opcion;

public static void main(String[] args) throws NumberFormatException, IOException {

do{

System.out.println("\n================MENU================\n");

System.out.println(" 1.- Vegetariano \n");

System.out.println(" 2.- No Vegetariano \n");

System.out.println(" 3.- Comida China \n");

System.out.println(" 4.- Salir \n");

System.out.println(" Ingrese su Opcion: ");

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

opcion = Integer.parseInt(br.readLine());

switch (opcion) {

case 1:{

Vegetariano vf = new Vegetariano();

System.out.println(vf.PrepararComida());

System.out.println(vf.PrecioComida());

} break;

case 2:{

Comida fl = new NoVegetariano((Comida) new Vegetariano());

System.out.println(fl.PrepararComida());

System.out.println(fl.PrecioComida());

} break;

case 3:{

Comida f2 = new ComidaChina((Comida) new Vegetariano());

System.out.println(f2.PrepararComida());

System.out.println(f2.PrecioComida());

} break;

default:{

System.out.println("\nHa salido del sistema\n");

}

return;

}

}while(opcion!=4);

}

}

public class NoVegetariano extends ComidaDecorator {

public NoVegetariano(Comida newComida){

super(newComida);

}

public String PrepararComida(){

return super.PrepararComida() + " Agregar Pollo ";

}

public double PrecioComida(){

return super.PrecioComida() +150.0;

}

}

public class ComidaChina extends ComidaDecorator {

public ComidaChina(Comida newComida) {

super(newComida);

}

public String PrepararComida() {

return super.PrepararComida() + " Agregar Arroz ";

}

public double PrecioComida() {

return super.PrecioComida() + 650.0;

}

}

El patrón en el proyecto elegido se representa de tal manera que, a la clase especificada para ser el decorador, extiende el comportamiento del objeto Comida, sin generar otras subclases. Los objetos que se desean crear manejan una estructura muy similar que hace pensar que son los mismos, aunque estos no sean los mismos. En el menú se pregunta al usuario la opción deseada y luego se llama al objeto Comida el cual es la interfaz del decorador, sin importar la opción elegida el decorador toma el objeto envuelto que se está solicitando para luego crearlo y guardarlo en los datos.

Este patrón funciona en este proyecto debido a que realiza la intención enfocada a crear objetos consecutivos al momento de la ejecución apilándolos sin tener que expandir en gran medida el código generando muchas opciones para combinar. Al encontrar que la clase ComidaChina y NoVegetariano guardan ciertas similitudes en las características de su implementación, se crea el decorador que permite añadirle funcionalidades al objeto Comida dependiendo de la opción requerida, tomando otros objetos que surjan del objeto principal.

A pesar de la buena implementación del patrón y de su funcionalidad, esto también puede tener algunas repercusiones negativas sobre el proyecto, quizá la más importante puede ser la dependencia del comportamiento del decorador debido al orden generado en la pila de los decoradores. Así mismo, para esta pila es complejo borrar un decorador especifico que hayamos generado. Esto quiere decir que, si queremos eliminar de la pila un decorador de la ComidaChina o NoVegetariano, no podremos eliminarlo en caso de que se encuentre en un lugar intermedio en la pila.

Finalmente, como se mencionó anteriormente otra posible solución para este problema adquirido en el proyecto, podría ser crear varias clases que signifiquen la combinación de las opciones de diversas maneras, pero esto no se tendría como una solución óptima ya que se expande en una gran medida el volumen código.