PRÁCTICA 03 - Modelado y Programación

Equipo "Better Code Saul"

Integrantes

Nombre	No. de cuenta
Álcantara Estrada Kevin Isaac	319073799
Cureño Sánchez Misael	418002485
Hernández Páramo Elizabeth	319143209

Instrucciones de compilacion

Forma 1

1. Dirigirse al directorio `src` de la práctica
2. Compilar usando

```
javac -sourcepath . -d ../target/ -cp .:../lib/* ./main/java/p03/Main.java
```

3. Copiar los recursos usando `cp -r ./main/resources ../target/main/`
4. Dirigirse al directorio generado `cd ../target`
5. Ejecutar usando `java -cp .:../lib/* main.java.p03.Main`

Forma 2 (Linux)

1. Ejecuta el script haciendo `./run.sh` o bien `bash run.sh` desde la terminal, dentro de la carpeta raiz de la práctica (en caso de no tener permiso de ejecución ejecutar `chmod +x run.sh`).

Instrucciones de formato estandarizado

1. En la raíz de la práctica ejecutar `make`.
2. En caso de que solo se quiera saber los errores de formato ejecutar `make check-format`.
3. En caso de que se quieran corregir los errores de formato ejecutar `make reformat`.

SECCIÓN TEÓRICA

Menciona los principios de diseño esenciales de los patrones Decorator y Adapter. Menciona una desventaja de cada patrón

DECORATOR

Como vimos en clase, este aptrón permite añadir funcionalidad a un objeto existente sin alterar su estructura. Los decoradores actúan como envolturas para los objetos de la clase original. Las envolturas proveen de funcionalidad adicional manteniendo su funcionalidad original intacta.

ESTRUCTURA

- **Componente:** Declara la interfaz común tanto para wrappers como para objetos envueltos.
- **Componente concreto:** Es una clase de objetos envueltos. Define el comportamiento básico, que los decoradores pueden alterar.
- **La clase Decoradora base:** Tiene un campo para referenciar un objeto envuelto. El tipo del campo debe declararse como la interfaz del componente para que pueda contener tanto los componentes concretos como los decoradores.
- **Decoradores concretos:** Definen funcionalidades adicionales que se pueden añadir dinámicamente a los componentes.
- **Cliente:** Puede envolver componentes en varias capas de decoradores, siempre y cuando trabajen con todos los objetos a través de la interfaz del componente.

IMPLEMENTACIÓN

1. Nos aseguramos de que tu dominio de negocio puede representarse como un componente primario con varias capas opcionales encima.
2. Decidimos ué métodos son comunes al componente primario y las capas opcionales. Creamos una interfaz de componente y declara esos métodos en ella.
3. Creamos una clase concreta de componente y define en ella el comportamiento base.
4. Creamos una clase base decoradora. Debe tener un campo para almacenar una referencia a un objeto envuelto. El campo debe declararse con el tipo de interfaz de componente para permitir la vinculación a componentes concretos, así como a decoradores.
5. Aseguramos de que todas las clases implementan la interfaz de componente.
6. Creamos decoradores concretos extendiéndolos a partir de la decoradora base. Un decorador concreto debe ejecutar su comportamiento antes o después de la llamada al método padre (que

siempre delega al objeto envuelto).
7. El código cliente debe ser responsable de crear decoradores y componerlos del modo que el cliente necesite.

DESVENTAJA

- Es difícil implementar un decorador de tal forma que su comportamiento no dependa del orden en la pila de decoradores.
- El código de configuración inicial de las capas pueden tener un aspecto desagradable.

ADAPTER

Como se nos explicó en clase, este patrón convierte la interfaz de una clase en otra interfaz que se adapte a la que el cliente espera. Permite a las clases trabajar juntas, a pesar de que sus interfaces sean incompatibles.

ESTRUCTURA

- **Clase cliente:** Contiene la lógica de negocio existente del programa.
- **Interfaz con el cliente:** Describe un protocolo que otras clases deben seguir para poder colaborar con el código cliente.
- **Servicio:** Es alguna clase útil (normalmente de una tercera parte o heredada). El cliente no puede utilizar directamente esta clase porque tiene una interfaz incompatible.
- **La clase Adaptadora:** Es capaz de trabajar tanto con la clase cliente como con la clase de servicio: implementa la interfaz con el cliente, mientras envuelve el objeto de la clase de servicio.

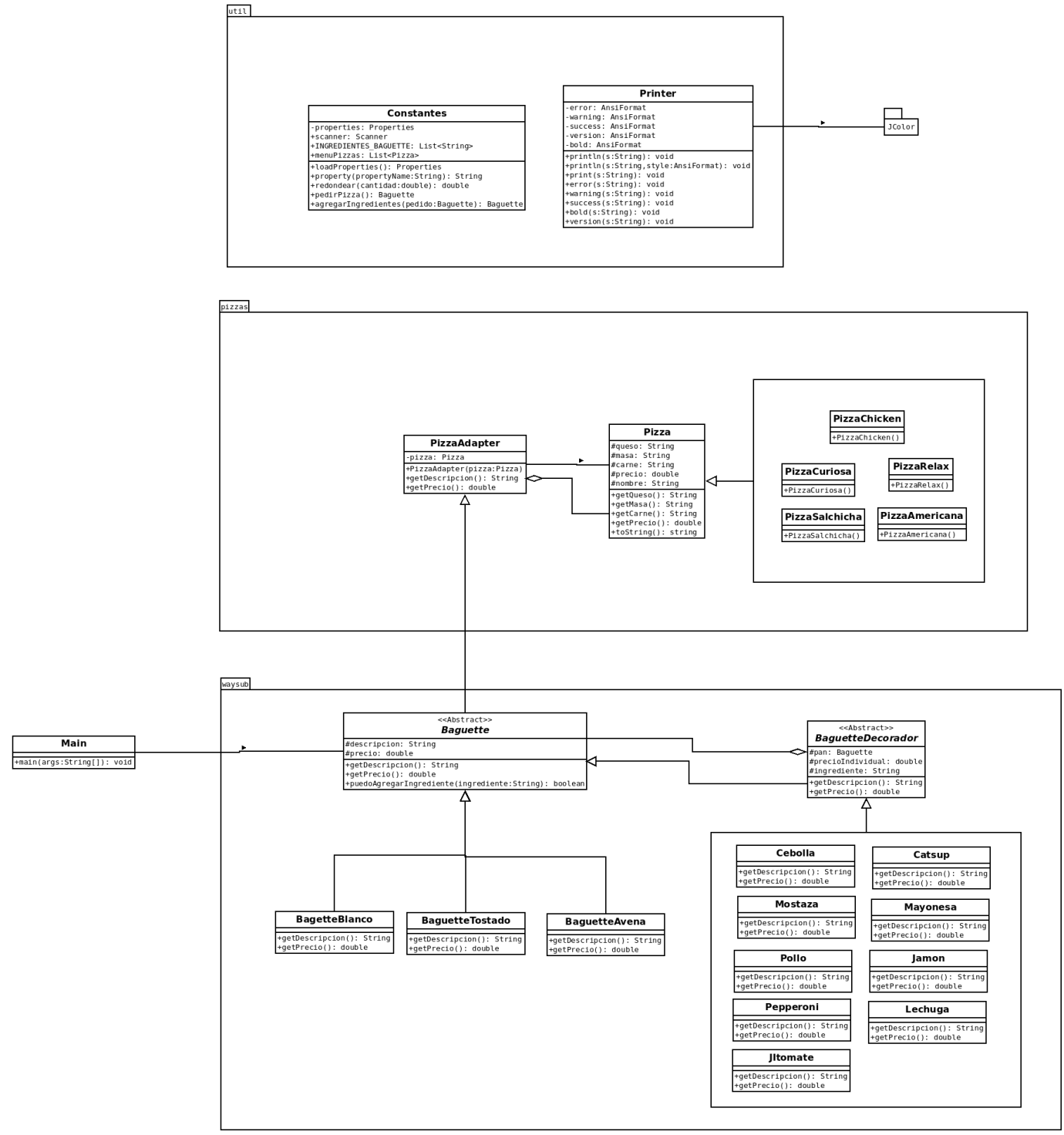
IMPLEMENTACIÓN

1. Nos aseguramos de que tienes al menos dos clases con interfaces incompatibles.
2. Declaramos la interfaz con el cliente y describimos el modo en que las clases cliente se comunican con la clase de servicio.
3. Creamos la clase adaptadora y haz que siga la interfaz con el cliente. Deja todos los métodos vacíos por ahora.
4. Añadimos un campo a la clase adaptadora para almacenar una referencia al objeto de servicio.
5. Implementamos todos los métodos de la interfaz con el cliente en la clase adaptadora.
6. Las clases cliente deberán utilizar la clase adaptadora a través de la interfaz con el cliente.

DESVENTAJA

- La complejidad general del código aumenta, ya que debemos introducir un grupo de nuevas interfaces y clases. En ocasiones resulta más sencillo cambiar la clase de servicio de modo que coincida con el resto del código.

Diagramas


util
Constantes
-properties: Properties
+scanner: Scanner
+INGREDIENTES_BAGUETTE: List<String>
+menuPizzas: List<Pizza>
+loadProperties(): Properties
+property(propertyName:String): String
+redondear(cantidad:double): double
+pedirPizza(): Baguette
+agregarIngredientes(pedido:Baguette): Baguette
Printer
-error: AnsiFormat
-warning: AnsiFormat
-success: AnsiFormat
-version: AnsiFormat
-bold: AnsiFormat
+println(s:String): void
+println(s:String,style:AnsiFormat): void
+print(s:String): void
+error(s:String): void
+warning(s:String): void
+success(s:String): void
+bold(s:String): void
+version(s:String): void
JColor
pizzas
PizzaAdapter
-pizza: Pizza
+PizzaAdapter(pizza:Pizza)
+getDescripcion(): String
+getPrecio(): double
Pizza
#queso: String
#masa: String
#carne: String
#precio: double
#nombre: String
+getQueso(): String
+getMasa(): String
+getCarne(): String
+getPrecio(): double
+toString(): string
PizzaChicken
+PizzaChicken()
PizzaCuriosa
+PizzaCuriosa()
PizzaRelax
+PizzaRelax()
PizzaSalchicha
+PizzaSalchicha()
PizzaAmericana
+PizzaAmericana()
waysub
Main
+main(args:String[]): void
<<Abstract>>
Baguette
#descripcion: String
#precio: double
+getDescripcion(): String
+getPrecio(): double
+puedoAgregarIngrediente(ingrediente:String): boolean
<<Abstract>>
BaguetteDecorador
#pan: Baguette
#precioIndividual: double
#ingrediente: String
+getDescripcion(): String
+getPrecio(): double
BagetteBlanco
+getDescripcion(): String
+getPrecio(): double
BaguetteTostado
+getDescripcion(): String
+getPrecio(): double
BaguetteAvena
+getDescripcion(): String
+getPrecio(): double
Cebolla
+getDescripcion(): String
+getPrecio(): double
Catsup
+getDescripcion(): String
+getPrecio(): double
Mostaza
+getDescripcion(): String
+getPrecio(): double
Mayonesa
+getDescripcion(): String
+getPrecio(): double
Pollo
+getDescripcion(): String
+getPrecio(): double
Jamon
+getDescripcion(): String
+getPrecio(): double
Pepperoni
+getDescripcion(): String
+getPrecio(): double
Lechuga
+getDescripcion(): String
+getPrecio(): double
JItomate
+getDescripcion(): String
+getPrecio(): double