

PRÁCTICA 1: RESOLUCIÓN DE PROBLEMAS



INSTITUTO POLITÉCNICO NACIONAL



"ESCOM" (ESCUELA SUPERIOR DE CÓMPUTO)

DOCENTE: MIGUEL ANGEL RODRIGUEZ CASTILLO

INTEGRANTES:

VÁZQUEZ BLANCAS CÉSAR SAID

MENDOZA SEGURA FERNANDO

U.A.: PARADIGMAS DE PROGRAMACIÓN

GRUPO: 3CM5

PROYECTO FINAL: RESTAURANTE

Proyecto final.

Planteamiento del problema

Descripción del problema: Aplicar los conocimientos adquiridos en Paradigmas de Programación para el desarrollo de programas en Lenguaje Java, con la finalidad de utilizar buenas prácticas en el diseño, implementación, pruebas y depuración de programas..

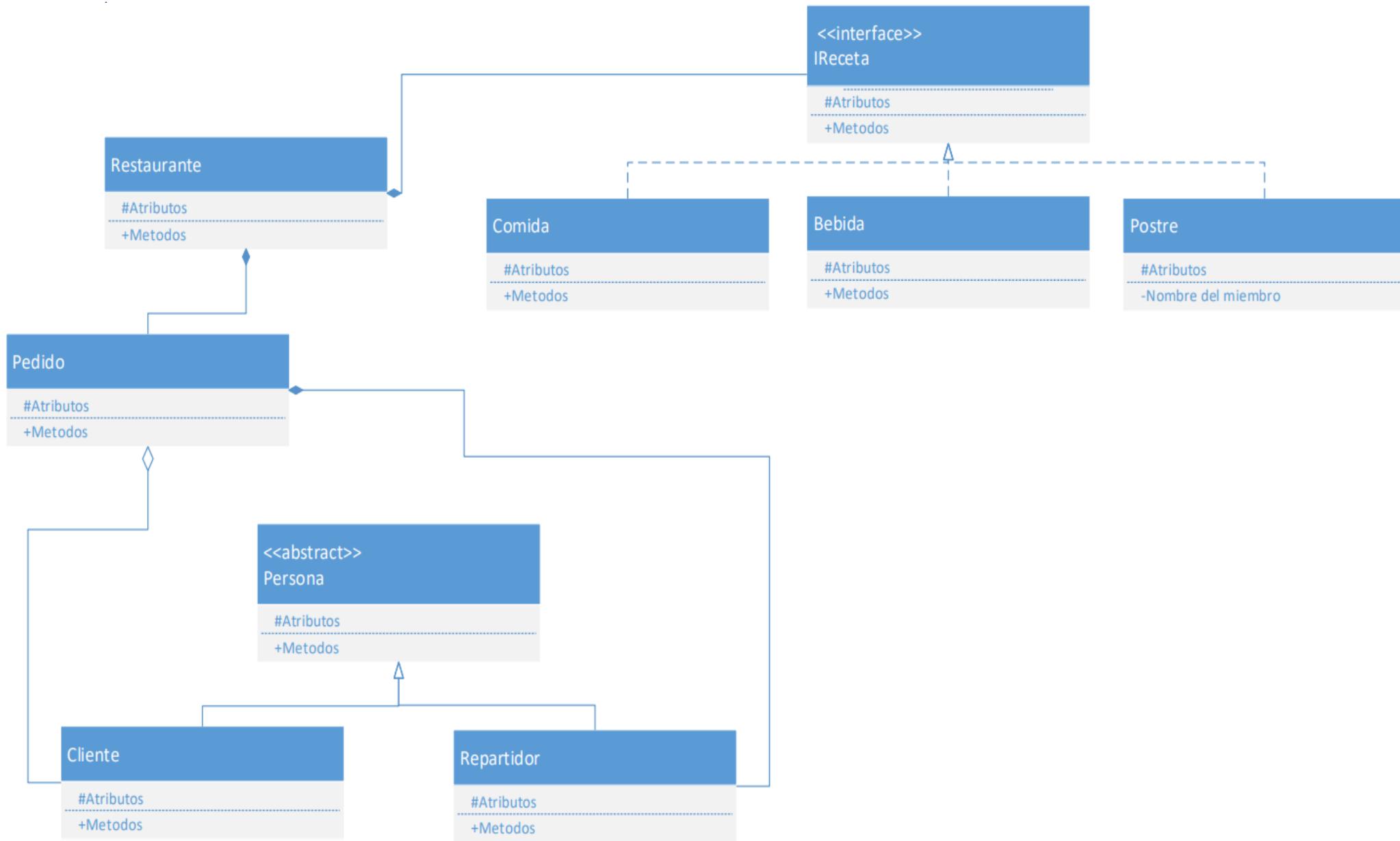
Entrada: Como datos de entrada tenemos múltiples variables las cuales serían nombres, direcciones, teléfonos, precios o elecciones, pero en general estamos utilizando como entrada datos de tipo String, int y double ya que son la información que proporcionamos al programa para comenzar su uso por ejemplo: para registrar un restaurante tenemos que proporcionar una dirección, su nombre y su número telefónico, para que después el programa haga uso de estos datos y los procese para después mostrarlo en el menú de la aplicación. También podemos irnos al caso de realizar un pedido en el que recibimos como entrada una variable String "elección" la cual será comparada con el nombre de alguno de los platos registrados previamente para verificar si el pedido se puede realizar o no ya que el plato no existe debido a que no recibió previamente una entrada "nombre" que sea igual a la "elección".

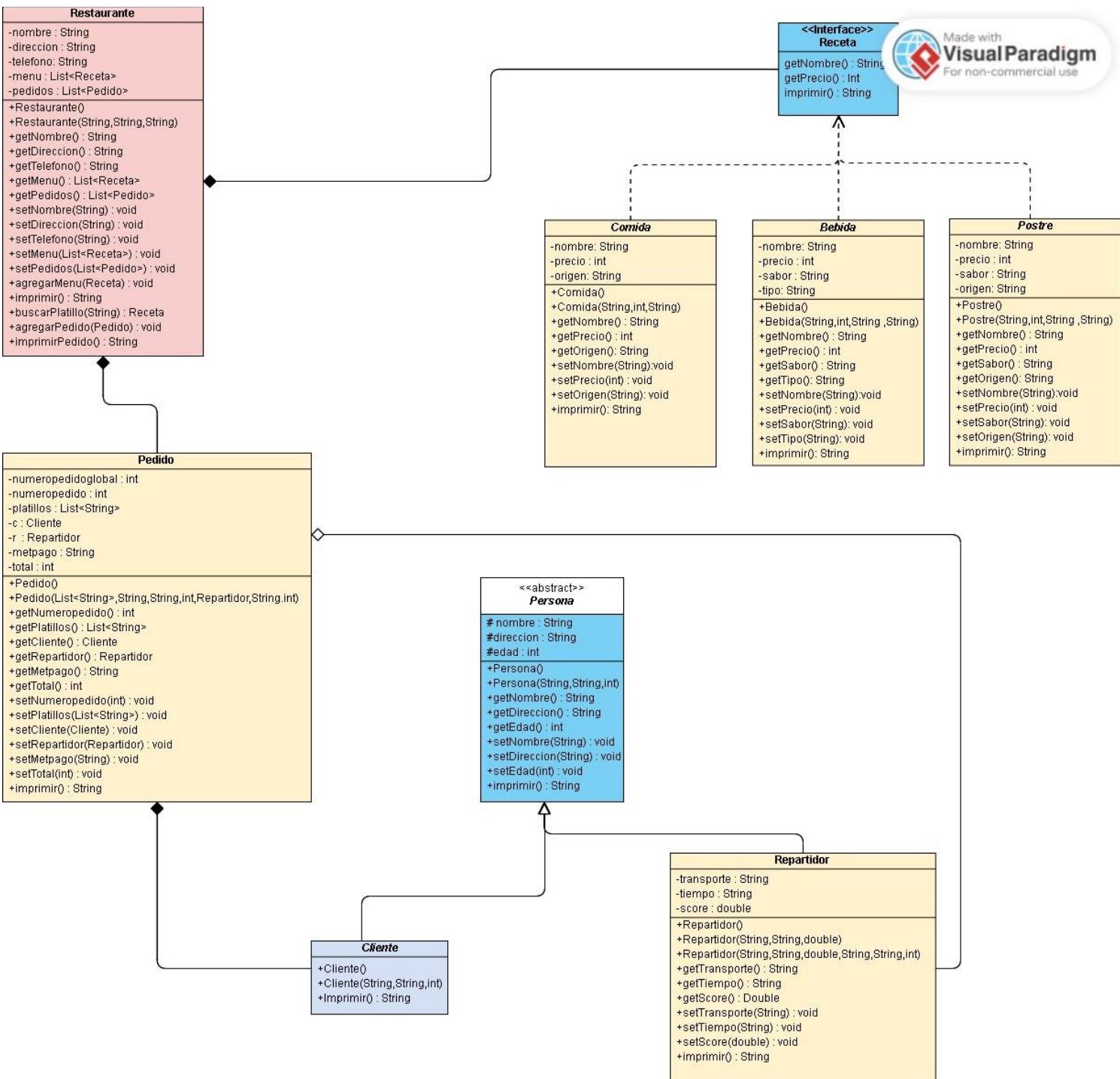
Proceso: El procesamiento se detalla en la implementación de la solución, pero en términos generales se define el comportamiento principal mediante la codificación del diagrama UML y después se añaden las funciones a la interfaz gráfica para interactuar con el código y orientar la aplicación a eventos. El proceso consiste en registrar un restaurante, luego múltiples platos, verificamos el registro en el botón del menú y finalmente registramos algunos repartidores para terminar realizando un pedido. Las clases definen el proceso lógico y los forms definen el proceso gráfico por lo que también recurrimos al uso de paquetes para poder importar nuestras clases al código que se encarga del aspecto gráfico y así poder interactuar con las instrucciones ya definidas.

Salida: Como primera salida tenemos los datos mostrados en el menú de un restaurante ya registrado siendo estos los platos que se ingresaron con sus precios, nombres, sabores, etc. También como salida está el nombre, dirección y teléfono registrador del restaurante. Como salidas finales tenemos la muestra de los datos de los repartidores registrados y la realización de un pedido con su respectiva información siendo esta los platos que se ordenaron, el repartidor que se le asignó, el total a pagar, el nombre del cliente, el tiempo de espera y la calificación score del repartidor.

Diseño y funcionamiento de la solución

Diagrama UML (Unified Modeling Language)





Implementación de la solución

Código desarrollado en lenguaje de programación Clases de comida, Clase “Postre”

Antes de empezar es necesario aclarar que primero explicaremos el funcionamiento del código descrito en java exceptuando la parte grafica ya que esa se detallara más a fondo después de entender la construcción de las clases que se muestran en el diagrama UML. Cabe mencionar que seguiremos la explicación según el diagrama yendo de arriba hacia abajo en cada clase, comenzando por las clases que refieren a un alimento.

Como primera declaración tenemos la declaración de un package de nombre modelo el cual nos servirá para agrupar todas las clases que estaremos creando en un solo paquete para su posterior uso. Como pudimos observar la clase postre es parte de la clase interfaz receta por lo que para poder indicar este hecho debemos hacer uso de la palabra reservada “implementes”.

Comenzamos declarando la clase y su visibilidad de la siguiente manera “public class Postre” y después escribimos la palabra reservada “implements” y la clase que va implementar, es decir, “Receta”. La declaración completa quedaría de la siguiente manera “public class Postre implements Receta {}.

```
package modelo;

public class Postre implements Receta{//herencia multiple o interfaz de receta
    private String nombre;
    private int precio;
    private String sabor;
    private String origen;

    public Postre() {//constructor vacio
    }

    public Postre(String nombre, int precio, String sabor, String origen) {//constructor con todos los atributos
        this.nombre = nombre;
        this.precio = precio;
        this.sabor = sabor;
        this.origen = origen;
    }
    @Override
    public String getNombre() {
        return nombre;
    }
    @Override
    public int getPrecio() {
        return precio;
    }
}
```

Una vez hecha la clase postre declaramos los atributos juntos con sus tipos de variable, siendo estos el nombre, precio, sabor y origen del postre.

Luego pasamos a la definición de los constructores por lo que de primera mano hacemos el constructor vacío el cual no tiene nada en su definición y el constructor básico que se declara primero colocando la visibilidad después el nombre de la clase y como parámetros colocamos los atributos de dicha clase, cabe mencionar que dentro de la definición instanciamos los atributos mediante el this.nombre = nombre, this.precio = precio, this.sabor = sabor, etcétera.

Ya hechos los constructores pasamos a la declaración de los getters y setters de cada uno de los atributos en la clase, a su vez definimos el método imprimir. En cuanto a los métodos “get” sabemos que su construcción es primero hacer la visibilidad luego el tipo de retorno y finalmente el get(Nombre del método) ya en su definición colocamos solamente el return seguido del nombre de dicha variable que se va a obtener. Respecto a los métodos get, su construcción inicia con su tipo de visibilidad luego un void y después el “set(Nombre del método)” ya en su definición colocamos el this.(nombre de la variable) = nombre de la variable a la que se está asignando un valor o accediendo.

Ahora explicando el método “public String imprimir()” comenzamos mostrando como es que se codifico.

```
@Override  
public String imprimir(){//método para imprimir todos los atributos  
    StringBuilder resultado = new StringBuilder(); //se construye una gran cadena para devolver y así se vea reflejada en el teclado  
    resultado.append("Nombre: ").append(nombre).append("\t"); //se construye la megacadena con append  
    resultado.append("Precio: ").append(precio).append("\t");  
    resultado.append("Sabor: ").append(sabor).append("\t");  
    resultado.append("Origen: ").append(origen).append("\t");  
    return resultado.toString(); //retornamos la cadena  
}
```

1.-“public String imprimir()”: Esta línea declara un método llamado imprimir que devuelve un objeto de tipo String. El modificador de acceso public indica que el método es accesible desde fuera de la clase.

2.-“StringBuilder resultado = new StringBuilder();” : Se crea una instancia de la clase StringBuilder. StringBuilder se utiliza para construir cadenas de manera eficiente, especialmente cuando se realizan múltiples concatenaciones de cadenas.

StringBuilder se usa para mejorar la eficiencia en la construcción de cadenas cuando necesitas realizar muchas operaciones de concatenación o modificaciones en el contenido de la cadena.

3.-resultado.append("Nombre: ").append(nombre).append("\t"); Aquí se agrega a la cadena en construcción (resultado) la información sobre el nombre del producto, seguido por un tabulador ("\t"). Se utiliza append para concatenar valores al StringBuilder.

Las líneas similares (resultado.append("Precio: ").append(precio).append("\t"), resultado.append("Sabor: ").append(sabor).append("\t"), resultado.append("Origen: ").append(origen).append("\t");) añaden al StringBuilder la información sobre el precio, sabor y origen del producto, respectivamente.

4.-“return resultado.toString();” : Convierte el StringBuilder en una cadena de tipo String y la devuelve. Este es el resultado final que contiene toda la información de los atributos concatenada en una sola cadena.

En resumen, este método se encarga de formatear la información de los atributos de un objeto (nombre, precio, sabor y origen) en una cadena y devolverla. Este tipo de métodos suele ser útil para imprimir información legible en la consola, en logs o en interfaces de usuario.

```

public String getsabor() {
    return sabor;
}

public String getorigen() {
    return origen;
}

public void setNombre(String nombre) {
    this.nombre = nombre;
}

public void setPrecio(int precio) {
    this.precio = precio;
}

public void setsabor(String sabor) {
    this.sabor = sabor;
}

public void setorigen(String origen) {
    this.origen = origen;
}

```

Los demás son la declaración de los getter y setters como se puede apreciar. Cabe mencionar que solo los métodos getNombre, getPrecio y el método public String imprimir(), hacen uso del @Override ya que la clase interfaz “Receta” solo hace uso de estos tres métodos, por lo que utilizamos el @Override para indicar la sobreescritura de dichos métodos.

CLASE “Bebida”

Cabe mencionar que como tal solo se esta haciendo uso de una clase interfaz y el implemento de dicha clase por lo que como tal no tenemos una herencia múltiple, pero si un concepto muy similar.

Para la clase Bebida nos basamos esencialmente en la misma construcción de la clase Postre ya que de igual manera implementa la interfaz Receta, por lo que la inicialización de la clase quedaría de la siguiente manera “public class Bebida implements Receta{}”.

Ya en la definición declaramos los atributos de la clase, siendo estos nombre, precio y origen, con sus respectivos tipos de datos.

Después definimos los constructores, tanto el vacío como el constructor base que hace uso de los atributos.

En cuanto a la declaración de los métodos solo se usa el @Override para los métodos get de Nombre, Precio y el método imprimir ya que son los que se implementan de la interfaz. La demás getters y setters se codifican con normalidad por lo que no hace falta detallar su estructura.

```
package modelo;

public class Comida implements Receta{//herencia multiple o interfaz de receta
    private String nombre;
    private int precio;
    private String origen;

    public Comida() {//constructor vacio
    }

    public Comida(String nombre, int precio, String origen) {//constructor con todos los atributos
        this.nombre = nombre;
        this.precio = precio;
        this.origen = origen;
    }
    @Override
    public String getNombre() {
        return nombre;
    }
    @Override
    public int getPrecio() {
        return precio;
    }

    @Override
    public String imprimir()//metodo para imprimir todos los atributos
    {
        StringBuilder resultado = new StringBuilder();
        resultado.append("Nombre: ").append(nombre).append("\t");//se construye una gran cadena para devolver y asi se vea re:
        resultado.append("Precio: ").append(precio).append("\t");//se construye la megacadena con append
        resultado.append("Origen: ").append(origen).append("\t");
        return resultado.toString();//retornamo la cadena
    }
    public String getOrigen() {
        return origen;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public void setPrecio(int precio) {
        this.precio = precio;
    }

    public void setOrigen(String origen) {
        this.origen = origen;
    }
}
```

La clase Comida es prácticamente igual a la clase Bebida y Postre ya que tiene como atributos el nombre, precio y origen, además de que hace uso de los mismos métodos, getters, setters y el método public String imprimir(). Cabe mencionar que la inicialización de la clase es la misma en esencia a las de Bebida y Postre, es decir que al implementar la clase Receta su declaración es “public class Comida implements Receta{}”, de igual forma se define su constructor vacío y su constructor normal.

```
package modelo;

public class Comida implements Receta{//herencia multiple o interfaz de receta
    private String nombre;
    private int precio;
    private String origen;

    public Comida() {//constructor vacio
    }

    public Comida(String nombre, int precio, String origen) {//constructor con todos los atributos
        this.nombre = nombre;
        this.precio = precio;
        this.origen = origen;
    }

    @Override
    public String getNombre() {
        return nombre;
    }

    @Override
    public int getPrecio() {
        return precio;
    }

    @Override
    public String imprimir(){//metodo para imprimir todos los atributos
        StringBuilder resultado = new StringBuilder();
        resultado.append("Nombre: ").append(nombre).append("\t");//se c
        resultado.append("Precio: ").append(precio).append("\t");//se c
        resultado.append("Origen: ").append(origen).append("\t");
        return resultado.toString();//retornamo la cadena
    }

    public String getOrigen() {
        return origen;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public void setPrecio(int precio) {
        this.precio = precio;
    }

    public void setOrigen(String origen) {
        this.origen = origen;
    }
}
```

CLASE “Receta”

```
package modelo;

public interface Receta {//interface receta
    public String getNombre(); //metodo general para devolver el nombre de comida,postre o bebida
    public int getPrecio(); //metodo general para devolver el precio de comida,postre o bebida
    public String imprimir(); //metodo general para imprimir dependiendo de que objeto sea, cada clase colocara lo que devolvera
}
```

Como podemos ver la clase Receta al ser la interfaz que se implementa en las demás tiene una declaración de acuerdo con su respectiva forma, es decir “public interface Receta{}” ya que Recta es la interfaz que contiene los métodos abstractos que se utilizan en las demás clases Postre, Bebida y Comida.

CLASE “Restaurante”

```
package modelo;
import java.util.ArrayList;
import java.util.List;
public class Restaurante {
    private String nombre;
    private String direccion;
    private String telefono;
    private List<Receta> menu;
    private List<Pedido> pedidos;

    public Restaurante() { //constructor vacio con las inicializaciones
        this.menu = new ArrayList<>(); //inicializamos la composicion receta
        this.pedidos=new ArrayList<>(); //inicializamos la composicion pedidos
    }

    public Restaurante(String nombre, String direccion, String telefono) { //constructor general
        this.nombre = nombre;
        this.direccion = direccion;
        this.telefono = telefono;
        this.menu = new ArrayList<>();
        this.pedidos=new ArrayList<>();
    }
}
```

Para la clase “Restaurante” debemos tener en claro que dicha clase tienen compuestas dos clases siendo estas la de “Receta” y “Pedido” por lo que tendremos uso se composición dentro de las definiciones de constructores o métodos en “Restaurante”.

De primera mano declaramos el paquete “modelo” y después importamos las librerías a utilizar las cuales son ArrayList y List mediante las líneas de código import java.util.ArrayList; import java.util.List;

import java.util.ArrayList;

Esta línea importa la clase ArrayList desde el paquete java.util. ArrayList es una implementación de la interfaz List en Java que proporciona una lista dinámica, lo que significa que su tamaño puede cambiar dinámicamente a medida que se agregan o eliminan elementos. En otras palabras, es un tipo de colección que puede almacenar elementos y que es muy eficiente para acceder a ellos por índice.

import java.util.List;

Esta línea importa la interfaz List desde el paquete java.util. List es una interfaz que representa una colección ordenada de elementos y forma parte del framework de colecciones en Java. La interfaz List define métodos para trabajar con listas, como agregar elementos, acceder a ellos por índice, obtener su tamaño, entre otros.

Usualmente, se utiliza List cuando se desea escribir código que trabaje con una lista de manera genérica, permitiendo que se utilice cualquier implementación específica de lista (por ejemplo, ArrayList, LinkedList, etc.).

La diferencia principal entre ArrayList y List en Java es que ArrayList es una implementación específica de la interfaz List.

Ya una vez declaradas las librerías necesarias para las operaciones que se van a realizar procedemos a la construcción de la clase Restaurante “public class Restaurante{}” junto con sus atributos los cuales serían el nombre (String), dirección (String) y teléfono (String). Los últimos dos atributos al ser parte de la composición tienen como tipo de dato los objetos que se componen.

Los atributos “menú” y “pedido” son listas que contiene como tipo de dato objetos los cuales son las clases compuestas respectivamente al diagrama de clases. La interfaz List representa una colección ordenada de elementos y permite el almacenamiento de elementos duplicados. La notación List<tipo de dato> es una forma de parametrizar la interfaz List con un tipo de dato específico. En este caso la lista “menu” contiene como tipo de dato un objeto de tipo “Receta”, en cuanto a la lista “pedidos” contiene como tipo de dato un objeto de tipo “Pedido”, de esta manera indicando parte de la composición de las clases Receta y Pedido.

Ya declarados los atributos, empezamos con la definición del constructor Restaurante, tanto el vacío como el que utiliza todos los atributos. Es necesario aclarar que desde el constructor vacío ambas listas se inicializan desde un principio mediante el “this.” Por ejemplo, en el caso de “menu” es

this.menu=new ArrayList<>(); , por ejemplo, en la lista de pedido : this.pedido=new ArrayList<>();

La expresión new ArrayList<>() en Java se utiliza para crear una nueva instancia de la clase ArrayList. En este caso, se utiliza la notación de diamante (<>), introducida en Java 7, para aprovechar la inferencia de tipo. La inferencia de tipo permite que el compilador determine automáticamente el tipo de datos adecuado en función del contexto.

Entonces, new ArrayList<>() está creando un nuevo objeto de tipo ArrayList, que es una implementación de la interfaz List. Esta instancia de ArrayList está destinada a almacenar elementos y tiene una

capacidad inicial predeterminada, que puede crecer automáticamente según sea necesario. La utilización de new ArrayList<>() es una forma concisa de instanciar un ArrayList sin tener que repetir el tipo de dato en ambos lados de la asignación, gracias a la inferencia de tipo.

Respecto al constructor normal la inicialización de los atributos es similar a la anteriores a excepción de las listas las cuales se inicializan mediante el “new ArrayList<>();”.

```
public String getNombre() {  
    return nombre;  
}  
  
public String getDireccion() {  
    return direccion;  
}  
  
public String getTelefono() {  
    return telefono;  
}  
  
public List<Receta> getMenu() {  
    return menu;  
}  
  
public List<Pedido> getPedidos() {  
    return pedidos;  
}  
  
public void setNombre(String nombre) {  
    this.nombre = nombre;  
}  
  
public void setDireccion(String direccion) {  
    this.direccion = direccion;|
```

```

public void setTelefono(String telefono) {
    this.telefono = telefono;
}

public void setMenu(List<Receta> menu) {
    this.menu = menu;
}

public void setPedidos(List<Pedido> pedidos) {
    this_pedidos = pedidos;
}

```

Luego de hacer los constructores, hacemos la codificación de todos los métodos getter y setter de los atributos de la clase “Restaurante”.

```

public String imprimir(){//metodo para imprimir el menu
    if (menu.isEmpty()) {//si no hay menu devolver la cadena no hay menu disponible
        return "No hay menú disponible";
    }
    StringBuilder resultado = new StringBuilder();//construccion de la megacadena
    for (Receta receta : menu) {//foreach para que todo lo que haya en la lista de receta se imprima
        if (receta instanceof Bebida bebida) {//si la instancia es bebida
            resultado.append(bebida.imprimir()).append("\n");//mandar a llamar a bebida.imprimir para imprimir sus atributos
        } else if (receta instanceof Comida comida) {//si la instancia es comida
            resultado.append(comida.imprimir()).append("\n");//mandar a llamar a comida.imprimir para imprimir sus atributos
        } else if (receta instanceof Postre postre) {//si la instancia es postre
            resultado.append(postre.imprimir()).append("\n");//mandar a llamar a postre.imprimir para imprimir sus atributos
        }
    }
    return resultado.toString(); //retornamos la mega cadena
}

```

Posteriormente pasamos a la codificación de la función imprimir la cual lleva un proceso mas complejo a anteriores funciones similares. De primera mano se tienen una sentencia condicional “if” la cual mediante el “.Empty” verifica si la lista “menu” esta vacía, por lo que si cumple con dicha condición se retorna el mensaje “No hay menú disponible”.

La parte “menu.isEmpty()” es una llamada a un método llamado isEmpty() que se aplica a un objeto llamado menu. Esta expresión se utiliza para verificar si la lista (o cualquier otra colección) denominada menu está vacía.

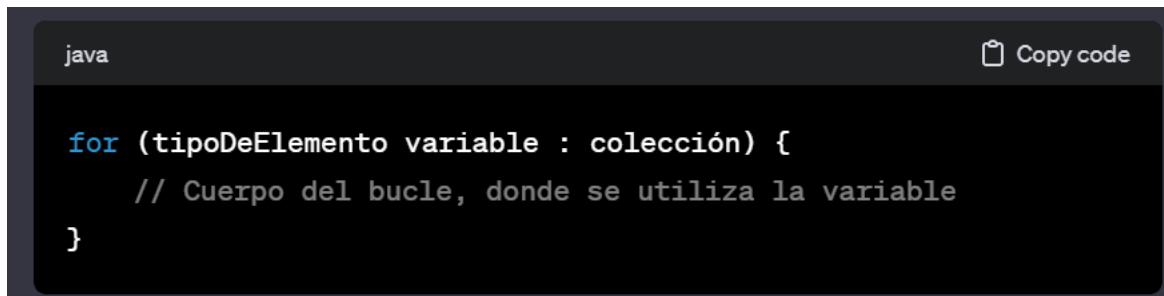
El método isEmpty() es comúnmente utilizado en Java para verificar si una colección, como una lista, conjunto o cadena, no contiene ningún elemento. Devuelve true si la colección está vacía y false si tiene al menos un elemento.

Si el “if” no se cumple se ejecuta la demás parte del código dentro de la función, es decir, si la lista menu no esta vacía, se pasa primero a crear una mega cadena de nombre “resultado” mediante el uso de StringBuilder.

Luego se hace uso de un “foreach”.

En Java, el bucle foreach se utiliza para iterar sobre elementos de una colección (como un array o una lista) sin necesidad de conocer la longitud de la colección ni de manipular un índice. Su sintaxis es más simple y fácil de leer que los bucles tradicionales.

La forma básica del bucle foreach en Java es la siguiente:



```
java
Copy code

for (tipoDeElemento variable : colección) {
    // Cuerpo del bucle, donde se utiliza la variable
}
```

-tipoDeElemento: Es el tipo de datos de los elementos contenidos en la colección. Por ejemplo, si estás iterando sobre un array de enteros, el tipo de elemento sería int. Si estás iterando sobre una lista de cadenas, el tipo de elemento sería String.

-variable: Es una variable que toma el valor de cada elemento en la colección en cada iteración del bucle.

-colección: Es la colección que estás recorriendo. Puede ser un array, una lista, un conjunto, etc.

El bucle para el código lleva como tipo de dato el objeto “Receta” y como variable “receta” siendo la colección el “menu”, o mas bien la lista que se esta recorriendo en la búsqueda. Se inicia un bucle foreach que itera sobre cada elemento de la lista menu, donde cada elemento es de tipo Receta. Se asume que menu es una colección de objetos de tipo Receta

Después dentro del foreach se tiene primero un condicional if el cual pregunta “if(receta instanceof Bebida bebida)” Se utiliza el operador instanceof para verificar si el objeto receta es una instancia de la clase Bebida. Si es así, se realiza un casting seguro (introducido en Java 14) para convertir receta en una variable local llamada bebida de tipo Bebida.

En Java, el operador instanceof se utiliza para verificar si un objeto es una instancia de una determinada clase, una instancia de una subclase o una instancia de una clase que implementa una interfaz específica. Proporciona una forma de comprobar la relación de herencia entre un objeto y una clase o interfaz.

La sintaxis básica es la siguiente:

java

 Copy code

```
objeto instanceof Tipo
```

Donde `objeto` es la instancia que estás verificando y `Tipo` es la clase o interfaz con la que estás comparando.

El uso de instanceof es común en situaciones donde se trabaja con herencia y polimorfismo, permitiendo tomar decisiones basadas en el tipo real de un objeto en tiempo de ejecución. Sin embargo, su uso excesivo a menudo indica un diseño que podría mejorarse mediante el uso de polimorfismo de manera más efectiva.

Posteriormente de cumplirse la condición del if, se ejecuta “resultado.append(bebida.imprimir()).append("\n);” para concatenar todos los atributos correspondientes en la megacadena “resultado” agregando el operador “resultado.append”. Cabe mencionar que el append agrega lo que está dentro del paréntesis siendo esta la llamada al método imprimir del objeto bebida, es decir “bebida.imprimir()”. Los demás condicionales que siguen ejecutan el mismo procedimiento, pero para los diferentes tipos de objetos que se tienen (comida, bebida y postre) e igualmente siguen el proceso de concatenamiento en “resultado” cambiando el hecho de que se debe modificar para el uso exclusivo del objeto en cuestión.

-else if (receta instanceof Comida comida) {: verifica si el objeto receta es una instancia de la clase Comida. Si es así, realiza un casting seguro a una variable local llamada comida de tipo Comida.

-resultado.append(comida.imprimir()).append("\n);: Se llama al método imprimir de la clase Comida y se agrega el resultado al StringBuilder resultado, seguido por un salto de línea.

-else if (receta instanceof Postre postre) {: Similar a los pasos anteriores, verifica si el objeto receta es una instancia de la clase Postre, y realiza un casting seguro a una variable local llamada postre de tipo Postre.

-resultado.append(postre.imprimir()).append("\n);: Se llama al método imprimir de la clase Postre y se agrega el resultado al StringBuilder resultado, seguido por un salto de línea.

-return resultado.toString();: Finalmente, la función retorna la representación de cadena de resultado, que contiene todas las impresiones de las recetas concatenadas.

```

public Receta buscarPlatillo(String eleccion){//recibimos una string para buscar en la lista de recetas
    for(Receta receta:menu){ //recorremos con el foreach
        if(receta.getNombre().equals(eleccion)){//si lo que hay en la receta su nombre es igual a la eleccion
            return receta; //retornamos el objeto de tipo receta completo
        }
    }
    return null; //si no retornamos null
}

```

Ya finalizado el método de imprimir pasamos al método “buscarPlatillo”. El método en cuestión retornará un objeto de tipo “Receta” por lo que colocamos esto como un tipo de retorno en el método después de su visibilidad, de igual manera se coloca que recibe como parámetro un String de nombre “elección” para buscar en la lista de recetas, siendo “eleccion” una variable que se definirá por el usuario y recibirá el método como un parámetro.

En la definición del método “buscarPlatillo” tenemos de primera mano un foreach el cual itera sobre cada elemento de la lista “menu”, donde cada elemento es de tipo “Receta”, siendo “receta” la variable donde se van guardando los datos de la lista. Luego tenemos dentro del foreach un condicional if el cual pregunta si el nombre de la receta (receta.getNombre()) es igual a la “eleccion” mediante el método “.equals”, es decir, “(receta.getNombre().equals(elección))”.

En Java, el método .equals() se utiliza para comparar si dos objetos son iguales. Este método está definido en la clase Object, que es la clase base de todas las clases en Java. Sin embargo, muchas clases personalizadas sobrescriben este método para proporcionar una comparación significativa entre instancias de la misma clase.

Si el nombre de la receta es igual a la elección se retorna el objeto “recta” de tipo “Receta” completo, de lo contrario se retorna un “null”.

```

public void agregarPedido(Pedido p) { //metodo para agregar un pedido a la lista de pedidos
    pedidos.add(p); //agregamos al pedido con el metodo add
}

```

Pasando al siguiente método, “agregarPedido(Pedido p)”, lo comenzamos definiendo con su visibilidad public, luego su retorno void , el nombre del método y finalmente que recibe como parámetro una variable de nombre “p” de tipo “Pedido”. Dentro de la estructura del método utilizamos el método “.add” para agregar un pedido “p” a la lista “pedidos”, lo que se vería de la siguiente manera: pedidos.add(p);

```

public String imprimirPedido(){
    if (pedidos.isEmpty()) { //si pedidos esta vacia mandar una string que diga que no hay pedidos
        return "No hay pedidos";
    }
    StringBuilder resultado = new StringBuilder(); //construimos la megacadena
    for (Pedido pe : pedidos) { //recorremos los pedidos con el foreach
        resultado.append(pe.imprimir()).append("\n"); //concatenamos con la impresion de todos los atributos de todos los
    }
    return resultado.toString(); //retornamos la megacadena para ser usada en el textArea
}

```

Como ultimo método tenemos “public String imprimirPedido()”. El método retorna un String por lo que colocamos esto después de la visibilidad. Dentro de la estructura tenemos primero la ejecución de un condicional if el cual pregunta mediante el método isEmpty() si la lista pedidos está vacía, de ser cierto este hecho se retorna un String “No hay pedidos”. Si el if no se cumple y la lista pedidos no está vacía, se sigue con el resto del código por lo que se crea una megacadena de nombre “resultado” mediante el método StringBuilder(), después se pasa a un foreach el cual itera sobre cada elemento la lista “pedidos” de tipo “Pedido” siendo la variable “pe” la que guarda las iteraciones, ya en la definición de foreach tenemos el uso del método append() para concadenar la impresión de todos los atributos de todos los pedidos en la megacadena resultado utilizando su respectivo método de impresión.

Para cada “Pedido (pe)” en la lista, se llama al método imprimir de la clase Pedido. El resultado de pe.imprimir() se agrega al StringBuilder resultado, seguido por un salto de línea (“\n”).

```
“resultado.append(pe.imprimir()).append(“\n”);”
```

Finalmente, mediante el “return resultado.toString();” retornamos la megacadena. El método retorna la representación de cadena (String) de resultado. Esta cadena contendrá todas las impresiones de los pedidos, cada una en una línea separada por un salto de línea. Este sería el ultimo método dentro de la clase “Restaurante”.

Clase “Pedido”

Ahora es importante mencionar que para esta clase en particular tenemos dos relaciones las cuales son composición y agregación. Cabe mencionar que en el caso del diagrama UML mostrado para el proyecto final, la clase que se compone en la clase “Pedido” es la de “Repartidor” y la que se agrega es la de “Cliente” sin embargo para termino más prácticos se intercambiaron dichas relaciones con el siguiente motivo:

El principal motivo es por la optimización de código y lógica de programación ya que visto desde el punto de vista estricto de las relaciones entre clases, es mejor que el cliente sea composición, ya que no hay registro y si un pedido se elimina el cliente también , en cambio sí es el repartidor, se eliminaría un repartidor ,lo cual en el lógica es incorrecto poque debe de haber repartidores en el restaurante siempre por lo que el repartidor debe ser agregación ya que ya hay una lista la cual solo se obtendrá un objeto y se agrega , si se elimina el pedido, se elimina ese registro pero no el repartidor.

En cambio, si la clase “Repartidor” fuera composición, sería eliminar un repartidor por un pedido , lo cual está mal ya que el repartidor se supone que está contratado. Ya aclarado esto pasamos a explicar el código de la clase.

```
package modelo;
import java.util.List;

public class Pedido {
    private static int numeropedidoglobal=1;//inicializamos en 1 para llevar el conteo general de pedidos
    private int numeropedido;
    private List<String> platillos;
    private Cliente c;
    private Repartidor r;
    private String metpago;
    private int total;

    public Pedido() { //constructor vacío
    }

    public Pedido(List<String> platillos, String nombre, String dirección, int edad, Repartidor r, String metpago, int total) {
        this.numeropedido = numeropedidoglobal++; //cada vez que se hace el constructor se aumenta en 1 el pedido
        this.platillos = platillos;
        c = new Cliente(nombre, dirección, edad); //composición de cliente
        this.r = r; //agregación de repartidor
        this.metpago = metpago;
        this.total = total;
    }
}
```

De primera mano declaramos nuestro paquete “modelo” e importamos la librería `java.util.List`, luego pasamos a la declaración de la clase “Pedido” (`public class Pedido{};`), después declaramos todos los atributos siendo el primero un atributo entero estático “numeropedidoglobal” el cual es igual a 1 para llevar el conteo general de pedidos.

La segunda variable es un entero de nombre “numeropedido”, a tercera es una lista que contendrá datos de tipo `String` de nombre “platillos”, la cuarta es una variable de nombre “c” de tipo `Cliente`, la quinta es una variable de nombre “r” de tipo “Repartidor”, la sexta es un `String` de nombre “metpago” y la séptima variable es un entero de nombre “total”. Todas las anteriores variables tienen una visibilidad “private”.

Después pasamos a la definición del constructor vacío y el constructor base de la clase `Pedido`. Cabe destacar que el constructor base tiene dos particularidades, la primera es que la inicialización de la variable “numeropedidoglobal” tiene un incremento en 1 en su inicialización (`this.numeropedidoglobal = numeropedidoglobal++1`) debido a que cada vez que se hace el constructor se aumenta en 1 el pedido, o dicho de otra forma, cada vez que se crea un objeto `Pedido` estamos inicializándolo con su constructor por lo que al generar varios pedidos estamos generando sus respectivos constructores por lo que para poder llevar el conteo global de los pedidos hacemos que incremente la variable “numeropedidoglobal”.

La segunda particularidad es con respecto al atributo “c”, al ser parte de la clase compuesta “`Cliente`” , el objeto se debe inicializar dentro del constructor de la clase principal por lo que “c” se debe instanciar en el constructor de la clase “`Pedido`”, lo que quedaría de la siguiente forma.

```

public Pedido(List<String> platillos, String nombre, String direccion, int edad, Repartidor r, String metpago, int total) {
    this.numeropedido = numeropedidoglobal++;//cada vez que se hace el constructor se aumenta en 1 el pedido
    this.platillos = platillos;
    c = new Cliente(nombre, direccion, edad); //composicion de cliente
    this.r = r; //agregacion de repartidor
    this.metpago = metpago;
    this.total = total;
}

```

Posteriormente pasamos a la definición de los métodos getter y setter de la clase “Pedido”.

```

public int getNumeropedido() {
    return numeropedido;
}

public List<String> getPlatillos() {
    return platillos;
}

public Cliente getC() {
    return c;
}

public Repartidor getR() {
    return r;
}

public String getMetpago() {
    return metpago;
}

public int getTotal() {
    return total;
}

public void setNumeropedido(int numeropedido) {
    this.numeropedido = numeropedido;
}

public void setPlatillos(List<String> platillos) {
    this.platillos = platillos;
}

public void setC(Cliente c) {
    this.c = c;
}

public void setR(Repartidor r) {
    this.r = r;
}

public void setMetpago(String metpago) {
    this.metpago = metpago;
}

public void setTotal(int total) {
    this.total = total;
}

```

Como ultimo método en esta clase tenemos el método “imprimir()” que tiene un retorno de tipo String y tiene una visibilidad de tipo “public”.

```

public String imprimir() { //metodo para imprimir todos los atributos
    StringBuilder resultado = new StringBuilder(); //se construye una gran cadena para devolver y asi se vea reflejada en el resultado
    resultado.append("Numero de pedido: ").append(numeropedido).append("\t"); //se construye la megacadena con append
    resultado.append("Cliente: ").append(c.getNombre()).append("\t");
    resultado.append("Repartidor: ").append(r.getNombre()).append("\t");
    resultado.append("Metodo de pago: ").append(metpago).append("\t");
    for (String p : platillos) { //for each para imprimir todo lo que hay en platillos
        resultado.append("Platillo: ").append(p).append("\t");
    }
    resultado.append("Total: ").append(total).append("\t");
    return resultado.toString(); //retornamos la cadena
}

```

Dentro de la definición del método, utilizamos la misma sintaxis y lógica que en los demás métodos de impresión que se explicaron con anterioridad. Se crea una megacadena para que mediante el operador

“append()” vamos concadenando en “resultado” todos los valores deseados que en este caso son como primera variable el número del pedido (append(numeropedido)), después el nombre del cliente (append(c.getNombre())), el nombre del repartidor (append(r.getNombre())), el método de pago (metpago), y luego se entra a un foreach el cual itera sobre la lista de platillos que contiene datos de tipo String y usa como variable a “p”.

El foreach se usa para imprimir y concatenar mediante append() los platillos que se encuentren mediante la línea de código “resultado.append(“Platillo: ”).append(p).append(“\t”);”. Ya fuera del ciclo foreach concatenamos a la megacadena el total mediante la línea de código “resultado.append(“Total: ”).append(total).append(“\t”);”.

Finalmente retornamos la cadena resultado con el “return resultado.toString();”

Clase “Persona”

Para no perder un poco la noción debemos hacer énfasis en que ahora pasaremos a explicar la clase “Persona” debido a que dicha clase hereda dos clases hijas las cuales son “Cliente” y “Repartidor” las cuales se componen y agregan respectivamente a la clase “Pedido” por lo que necesitamos entender la clase “Persona” para después comprender con mayor facilidad a sus clases hijas.

La clase “Persona” se define como una clase abstracta según el diagrama UML por lo que primero declaramos la clase con su visibilidad, seguida de la palabra “abstract”, lo que quedaría de la siguiente forma: “public abstract class Persona{}”. Para que “Persona” sea una clase abstracta, aparte de cumplir con la anterior definición, también dicha clase debe integrar por lo menos un método abstracto el cual no tienen definido como realizar un proceso por lo que solo tiene en esencia el método definido mas no sus operaciones.

Luego se declaran los atributos de la clase, siendo estos nombres (String), dirección (String) y edad (int). Hechos los atributos se definen el constructor vacío y el constructor base. Posteriormente se estructuran los métodos getter y setter de la clase “Persona”. Cabe mencionar que la visibilidad de los atributos es de acuerdo con el diagrama por lo que tienen un acceso de tipo “protected”.

Finalmente definimos el método abstracto el cual en este caso sería “public abstract String imprimir()”. Este método no tiene más lógica ya que como tal solo estamos declarando un método que no tiene instrucciones ya que las clases hijas son las que integraran sus debidos pasos u operaciones a realizar.

```
package modelo;

public abstract class Persona {
    protected String nombre;
    protected String direccion;
    protected int edad;

    public Persona() {//constructor vacio
    }

    public Persona(String nombre, String direccion, int edad) {//constructor con todos los atributos
        this.nombre = nombre;
        this.direccion = direccion;
        this.edad = edad;
    }

    public String getNombre() {
        return nombre;
    }

    public String getDireccion() {
        return direccion;
    }

    public int getEdad() {
        return edad;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public void setDireccion(String direccion) {
        this.direccion = direccion;
    }

    public void setEdad(int edad) {
        this.edad = edad;
    }
}
```

```

    public void setEdad(int edad) {
        this.edad = edad;
    }

    public abstract String imprimir();
}

```

Clase “Cliente”

Esta clase es una subclase o clase hija de la cual hereda tanto sus atributos como métodos de la clase “Persona”. Su explicación es sumamente sencilla ya que comenzamos primero con la declaración del paquete “modelo” después pasamos a la declaración de la clase y agregamos después del nombre la palabra “extends” seguido del nombre de la clase que le dio la herencia, es decir, “Persona”.

“public class Cliente extends Persona{}”

Finalmente definimos los constructores, tanto el vacío como el constructor base. Es importante destacar que en cuanto a la inicialización del constructor base de cliente utilizaremos sus atributos y los que hereda, en este caso solo se hace uso de lo heredados ya que no se declararon atributos propios por lo que los escribimos dentro de los parámetros del constructor y en su definición los inicializamos mediante el operador “super()”, colocando dentro del paréntesis el nombre de los atributos que se heredan.

Recordemos que “super()” solo se utiliza en los atributos heredados por lo que, si alguna clase tiene también atributos propios, dichas variables se deben inicializar por aparte mediante el “this.(nombre) = (nombre)”.

También se colocar el método “public String imprimir(){}” heredado de la clase abstracta, pero se le coloca en su definición que retorna un “null”. Dicho método viene acompañado de la notación @Override al ser sobreescrito.

```

package modelo;

public class Cliente extends Persona{//herencia de persona

    public Cliente() {
    }

    public Cliente(String nombre, String direccion, int edad) {//constructor de cliente
        super(nombre, direccion, edad); //instanciamos los atributos de persona para cliente
    }
    @Override
    public String imprimir(){
        return null;
    }
}

```

Clase “Repartidor”

La clase “Repartidor” al igual que la clase “Cliente” son parte de la herencia de la clase “Persona” por lo que sigue los mismos principios de la utilización de “super” para la inicialización de los atributos heredados en su constructor, e igualmente el uso de la notación @Override para los métodos sobrescritos.

```
package modelo;

public class Repartidor extends Persona{//herencia de persona
    protected String transporte;
    protected String tiempo;
    protected double score;

    public Repartidor() {//constructor vacio
    }

    public Repartidor(String transporte, String tiempo, double score) {//constructor con todos los atributos de repartidor
        this.transporte = transporte;
        this.tiempo = tiempo;
        this.score = score;
    }

    public Repartidor(String transporte, String tiempo, double score, String nombre, String direccion, int edad) {//constructor con todos los atributos de repartidor y de persona
        super(nombre, direccion, edad); //se inicializa la herencia
        this.transporte = transporte;
        this.tiempo = tiempo;
        this.score = score;
    }
}
```

Como podemos apreciar primero se declara el paquete utilizado “modelo” y después se inicia la clase utilizando el extends para indicar la herencia (public class Repartidor extends Persona{}). Luego pasamos a la declaración de sus atributos: transporte (String) , tiempo (String) y score (double). Estos atributos tienen una visibilidad “protected”. Posteriormente pasamos a la definición del constructor vacío y el constructor base , utilizando en este ultimo la palabra super para inicializar los atributos heredados y el “this.(nombre de variable)=(nombre de variable)” para los atributos propios. Después pasamos a la construcción de los métodos getter y setter.

```

package modelo;

public class Repartidor extends Persona{//herencia de persona
    protected String transporte;
    protected String tiempo;
    protected double score;

}

public Repartidor() {//constructor vacio
}

public Repartidor(String transporte, String tiempo, double score) {//constructor con todos los atributos de repartidor
    this.transporte = transporte;
    this.tiempo = tiempo;
    this.score = score;
}

public Repartidor(String transporte, String tiempo, double score, String nombre, String direccion, int edad) {//constructor con todos los atributos de persona y repartidor
    super(nombre, direccion, edad); //se inicializa la herencia
    this.transporte = transporte;
    this.tiempo = tiempo;
    this.score = score;
}

public String getTransporte() {
    return transporte;
}

public String getTiempo() {
    return tiempo;
}

public double getScore() {
    return score;
}

public void setTransporte(String transporte) {
    this.transporte = transporte;
}

public void setTiempo(String tiempo) {
    this.tiempo = tiempo;
}

public void setScore(double score) {
    this.score = score;
}

```

Finalmente pasamos al método heredado de la clase abstracta “imprimir()” el cual sigue la misma lógica que todos los anteriores métodos similares pues se crea una megacadena mediante un StringBuilder y se van concadenando varios valores mediante el operador “append()” colocando dentro de los paréntesis los valores que deseamos agregar a la cadena para finalmente retornar por completo toda la cadena.

```

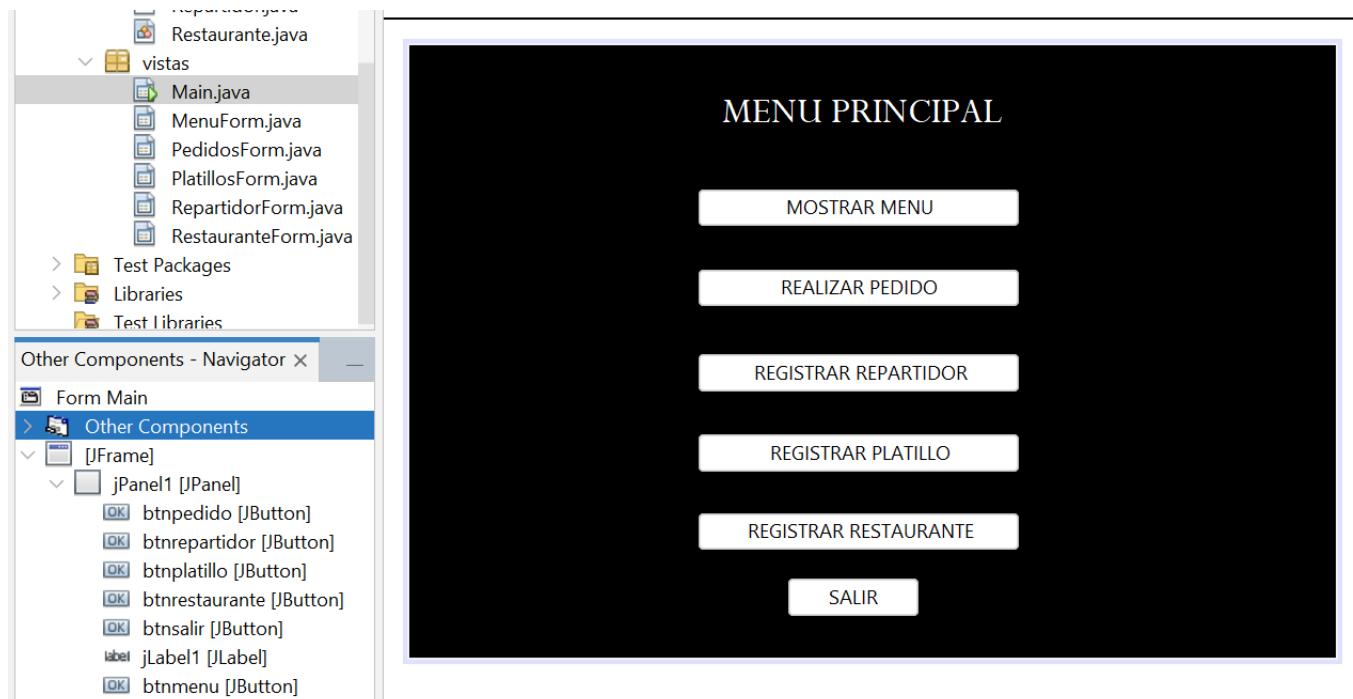
@Override
public String imprimir(){//metodo para imprimir todos los atributos
    StringBuilder resultado = new StringBuilder(); //se construye una gran cadena para devolver y asi se vea reflejada en el resultado.append("Nombre: ").append(nombre).append("\t");//se construye la megacadena con append
    resultado.append("Direccion: ").append(direccion).append("\t");
    resultado.append("Edad: ").append(edad).append("\t");
    resultado.append("Medio de transporte: ").append(transporte).append("\t");
    resultado.append("Tiempo de espera estimado: ").append(tiempo).append("\t");
    resultado.append("Score: ").append(score).append("\t");
    return resultado.toString(); //retornamos la cadena
}

```

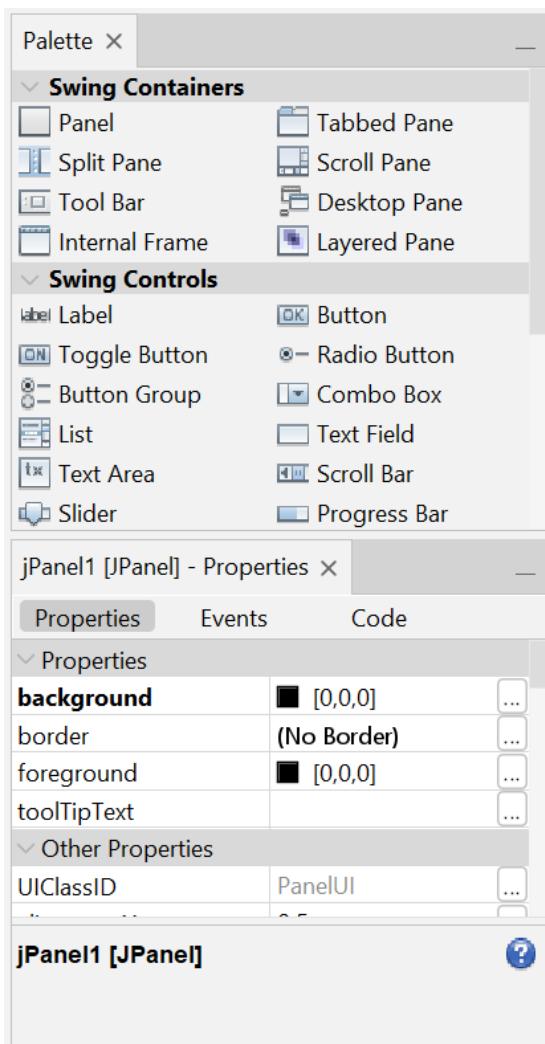
Interfaz gráfica.

Form Main ; Main :: JFrame

En este momento pasaremos a explicar en esencia como es que se estructuro la interfaz gráfica, comenzando por la primera pantalla que se muestra al ejecutar el programa, es decir, el Form Main.



Como podemos apreciar estamos en la pestaña de diseño del Form Main en el que podemos apreciar en la barra del lado izquierdo los componentes que se agregaron, siendo las variables de tipo [JButton] los botones que se encargarán de llevar a cabo una acción, es necesario destacar que cada uno tiene su respectivo nombre lo que permitirá identificarlos en el apartado de código. También hay otras dos variables, uno es el panel principal "JPanel1" de tipo [JPanel] el cual es el fondo que se puede apreciar de color negro, dicho parámetro se establece en las mismas opciones que nos proporciona NetBeans.



Podemos ver rápidamente la barra de opciones la cual contiene todas las opciones graficas que tenemos al alcance las cuales ya definen de que tipo son, por ejemplo, un "Panel" , una etiqueta de texto "Label" o un botón "Button".

Ahora pasando al otro componente que se agregó, es la etiqueta que contiene el texto de "Menu Principal". En general esta es la forma mas esencial en la que se maneja el diseño de la parte grafica por lo que no es necesario explicar muy a fondo el funcionamiento ya que es muy intuitivo su uso.

Ya explicado el diseño del Form Main, pasaremos a la parte del código que es donde se relaciona el trabajo hecho con anterioridad del diagrama UML.

```
package vistas;
import javax.swing.JOptionPane;
import modelo.Restaurante;
import java.util.List;
import java.util.ArrayList;
import modelo.Repartidor;
import java.util.Random;
```

Pasando al código tenemos como primer paso declarar el paquete "vistas" el cual contendrá todos los Forms que estaremos utilizando para el proyecto. Luego importamos dos librerías del paquete "modelo" las cuales son "Restaurante" y "Repartidor". Posteriormente importamos diversas librerías que nos proporcionaran las herramientas necesarias para lo que resta de código e instrucciones:

-La biblioteca Swing para crear una interfaz gráfica de usuario (GUI). La línea de código import javax.swing.JOptionPane; importa la clase JOptionPane del paquete javax.swing. La clase JOptionPane es parte de la biblioteca Swing en Java y proporciona un conjunto de métodos para mostrar cuadros de diálogo modales que permiten la interacción con el usuario en aplicaciones de interfaz gráfica de usuario (GUI).

- La librería útil.Random sirve para generación de números aleatorios (Random).
- Las librerías List y ArrayList son para trabajar sobre listas y operaciones que se pueden ejecutar sobre las mismas.

Ahora pasamos a la estructura de la clase "Main".

```

public class Main extends javax.swing.JFrame {
    private Restaurante r; //inicializamos una clase restaurante para guardar ahí los datos
    private List<Repartidor> rep; //inicializamos una lista de repartidores para registrarlos y después tomarla como referencia

    public Main() {
        initComponents();
        r=new Restaurante(); //inicializamos cada vez que se corre el proyecto
        rep=new ArrayList<>(); //inicializamos cada vez que se corre el proyecto
        this.setLocationRelativeTo(null); //colocamos al centro el form main
    }

    @SuppressWarnings("unchecked")
    // Generated Code

```

Primero la línea “public class Main extends javax.swing.JFrame {“ muestra cómo se declara una clase llamada Main que extiende de javax.swing.JFrame. Esto significa que la clase Main es una ventana (frame) de Swing. Todas las funcionalidades y características de una ventana Swing estarán disponibles en esta clase.

Después se declaran dos atributos privados siendo el primero de tipo “Restaurante” de nombre “r” y el segundo de tipo List<> que integra datos de tipo “Restaurante” (List<Repartidor>) la cual tiene como nombre “rep”. La clase Restaurante inicializada sirve para guardar dentro los datos y la lista de datos tipo “Repartidor”, rep, sirve para registrar a los repartidores, llevar como referencia en la búsqueda y asignación de un pedido.

Luego en el constructor de la clase Main (public Main(){}), se realizan las siguientes operaciones:

-Se manda a llamar al método initComponents(); El método initComponents() es generado automáticamente por NetBeans en proyectos Swing cuando utilizas el diseñador gráfico de interfaz de usuario de NetBeans (GUI Builder) para diseñar tus formularios. Este método se utiliza para inicializar y configurar todos los componentes visuales que has arrastrado y colocado en tu formulario.

Cuando diseñas un formulario en NetBeans, el diseñador gráfico genera automáticamente el código necesario para la interfaz gráfica, y este código suele incluir una llamada a initComponents(). Aquí hay algunas cosas que realiza este método:

1.-Inicialización de Componentes: Configura y inicializa todos los componentes visuales (botones, etiquetas, campos de texto, etc.) que has agregado al formulario. Esta inicialización incluye la creación de instancias de los objetos y la configuración de sus propiedades.

2.-Distribución de Diseño: Si has utilizado el diseñador para organizar visualmente tus componentes en el formulario, este método también puede contener el código que establece la disposición (layout) de los componentes en el contenedor principal.

3.-Manejo de Eventos: Si has asociado eventos a componentes en el diseñador, como hacer clic en un botón, es probable que parte del código de manejo de eventos también se genere dentro de initComponents().

4.-Código Generado: NetBeans agrega automáticamente el código generado para cada componente visual al método initComponents(). Este código puede ser extenso y detallado, pero se oculta en este método para mantener el código principal más limpio y centrado en la lógica de la aplicación.

-Inicializamos la variable “r” creando una nueva instancia de la clase “Restaurante” . Al estar en el constructor la acción ocurre cada que se corre el proyecto.

-Inicializamos la lista de repartidores “rep” creando una nueva instancia de “ArrayList”, lo que se vería de la siguiente forma : rep = new ArrayList<>(); Cae mencionar que al estar dentro del constructor esta acción, ocurre lo mismo que pasa en la inicialización de “r” cada vez que se corre el proyecto.

-Finalmente mediante la línea “this.setLocationRelativeTo(null)” estamos colocando la ventana grafica en el centro de la pantalla.

Después del constructor viene una línea de código “@SuppressWarnings(“unchecked”)” . Esta anotación indica al compilador que ignore ciertas advertencias generadas por el código. En este caso, se utiliza para suprimir advertencias relacionadas con la verificación de tipos en la creación de componentes de la interfaz gráfica.

Hecho esto pasamos al código generado gracias al método initComponents();

```
private void btnmenuActionPerformed(java.awt.event.ActionEvent evt) {  
    if(r.getNombre()==null){//si no hay restaurante , mandara un mensaje para que se registre  
        JOptionPane.showMessageDialog(null, "No hay restaurante registrado", "Información", JOptionPane.INFORMATION_MESSAGE);  
    }  
    else{//si hay restaurante  
    setVisible(false); //ocultamos el main  
    MenuForm mf = new MenuForm(Main.this); //inicializamos el form del menu  
    mf.setVisible(true); //mostramos el frame del menu  
    }  
}
```

Como parte del código generado solo tenemos la inicialización de los métodos que están orientados a los eventos relacionados con la interfaz gráfica, es decir que por cada botón que definimos en el diseño se creó un método para poder manipularlos e indicarles un funcionamiento en particular. Todo esto se da mediante la línea:

“private void btnmenuActionPerformed(java.awt.event.ActionEvent evt) {“

En Java, especialmente en el contexto de interfaces gráficas de usuario (GUI) y eventos, java.awt.event.ActionEvent es una clase que representa un evento de acción. Este evento se genera cuando se realiza una acción en un componente de la interfaz gráfica, como hacer clic en un botón. La variable evt la cual es una instancia de ActionEvent y se utiliza para proporcionar información sobre el evento que ha ocurrido.

La firma del método indica que es un manejador de eventos para el botón llamado btnmenu. Cuando este botón se activa (por ejemplo, al hacer clic en él), se ejecuta el código dentro del método. La variable evt se usa para acceder a información sobre el evento de acción que ha tenido lugar.

Pasando al código del método lo explicaremos paso a paso:

1.-if(r.getNombre()==null){: Esta línea verifica si el nombre del restaurante (r.getNombre()) es nulo. Parece que r es una instancia de la clase Restaurante y se está verificando si el restaurante está registrado o no.

2.-JOptionPane.showMessageDialog(null, “No hay restaurante registrado”, “Información”, JOptionPane.INFORMATION_MESSAGE): Si el nombre del restaurante es nulo (no hay restaurante registrado), se muestra un cuadro de diálogo (JOptionPane.showMessageDialog) con un mensaje informativo indicando que no hay restaurante registrado. El parámetro null en este caso significa que el cuadro de diálogo se centrará en la pantalla. JOptionPane.INFORMATION_MESSAGE especifica el ícono que se mostrará en el cuadro de diálogo, indicando que es un mensaje informativo.

3.-else: Si hay un restaurante registrado (el nombre no es nulo), se ejecutan las instrucciones en el bloque else.

4.-setVisible(false): Oculta la ventana actual (Main). Esto parece ser un formulario principal (Main), y este comando oculta el formulario principal cuando se hace clic en el botón.

5.-MenuForm mf = new MenuForm(Main.this): Se crea una instancia de la clase MenuForm, que es un formulario para mostrar el menú. Se pasa la instancia actual de Main (Main.this) al constructor de MenuForm. Esto puede ser útil para establecer alguna relación o referencia entre los formularios.

6.-mf.setVisible(true): Hace visible el formulario del menú (MenuForm). Muestra el formulario del menú después de haber ocultado el formulario principal (Main).

La línea de código completa crea un nuevo objeto de la clase MenuForm y lo asigna a la variable mf. Esto proporciona una manera de acceder y manipular el formulario del menú (MenuForm) desde el código de la clase Main.

```
private void btnplatilloActionPerformed(java.awt.event.ActionEvent evt) {  
    if(r.getNombre()==null){//si no hay restaurante , mandara un mensaje para que se registre  
        JOptionPane.showMessageDialog(null, "No hay restaurante registrado", "Información", JOptionPane.INFORMATION_MESSAGE);  
    }  
    else{//si hay restaurante  
        setVisible(false); //ocultamos el main  
        PlatillosForm pf=new PlatillosForm(Main.this); //inicializamos el form de platillos  
        pf.setVisible(true); //mostramos el form de los platillos  
    }  
}
```

Para el método “btnplatilloActionPerformed” sigue la misma lógica del método anterior para el botón de menu, lo único que cambia es la inicialización del form en caso de haber restaurante ya que el ahora se inicializa una instancia de la clase “PlatillosForm” de nombre “pf”, este nombre sirve como un identificador que permite interactuar con el objeto PlatillosForm. “Main.this” Se utiliza en el contexto de una clase anidada (como una clase interna) para referirse a la instancia de la clase externa (Main en este caso). Puede ser necesario cuando se está dentro de una clase interna y se necesita referenciar la instancia de la clase externa.

```
private void btnsalirActionPerformed(java.awt.event.ActionEvent evt) {  
    System.exit(0); //el boton salir hara que se cierre el proyecto  
}
```

Para el método del botón Salir, tenemos “btnsalirActionPerformed” el cual sigue las mismas instrucciones para iniciar la estructura del método integrando como parámetro el “java.awt.event.ActionEvent evt”, lo único que cambia es la definición del método ya que solo integra la línea de código “System.exit(0);” que da a entender que se cerrara el proyecto al seleccionar el botón.

La instrucción System.exit(0) en Java se utiliza para finalizar la ejecución de una aplicación Java. La llamada a System.exit(0) indica que la aplicación debe terminar normalmente con un estado de salida específico. Aquí hay una explicación más detallada:

-System.exit(0):

-System.exit es un método estático de la clase System.

-Toma un argumento, que es el código de estado de salida. En este caso, 0 indica que la aplicación se está cerrando sin errores.

Cuando System.exit(0) se ejecuta, la máquina virtual de Java (JVM) se apaga y la aplicación termina. La elección del código de estado 0 es una convención para indicar que la aplicación se está cerrando normalmente sin errores. Otros códigos de salida, distintos de cero, pueden indicar diversos tipos de errores o condiciones específicas de salida.

En el contexto de NetBeans y otras IDEs (entornos de desarrollo integrados), la llamada a System.exit(0) a menudo se encuentra en el método main de la clase principal de la aplicación. Cuando ejecutas una aplicación desde un IDE y la aplicación termina (por ejemplo, al hacer clic en el botón de cierre de la ventana), el IDE puede cerrar la ejecución de la aplicación llamando a System.exit(0) para asegurarse de que todos los recursos se liberen adecuadamente antes de finalizar el programa.

```
private void btnrestauranteActionPerformed(java.awt.event.ActionEvent evt) {  
    setVisible(false); //ocultamos el main  
    RestauranteForm rf=new RestauranteForm(Main.this); //mostrara el form para registrar restaurante  
    rf.setVisible(true); //se muestra el form  
}
```

Para el botón de “Registrar Restaurante” que tiene como nombre en el código “btnrestauranteActionPerformed”, solo tiene la inicialización de la instancia de la clase “RestauranteForm” por lo que sigue la misma lógica de los anteriores métodos de botones de registro, pero esta vez sin el condicional if que pregunta si hay registrado un restaurante pues este botón sirve precisamente para registrar un restaurante.

RestauranteForm: Indica que se está creando una instancia de la clase RestauranteForm.

rf: rf es el nombre de la variable que se utiliza para hacer referencia a la instancia recién creada de RestauranteForm. Esta variable permite interactuar con el objeto RestauranteForm en el código.

= new RestauranteForm(Main.this): Aquí es donde se crea la instancia de RestauranteForm. new RestauranteForm() crea un nuevo objeto de la clase RestauranteForm. Main.this se pasa como argumento al constructor de RestauranteForm.

Entonces, la línea de código completa crea una nueva instancia de RestauranteForm y la asigna a la variable rf. Esto proporciona una manera de acceder y manipular el formulario del restaurante (RestauranteForm) desde el código de la clase Main.

```
private void btnpedidoActionPerformed(java.awt.event.ActionEvent evt) {  
    if(r.getNombre()==null){ //si no hay restaurante , mandara un mensaje para que se registre  
        JOptionPane.showMessageDialog(null, "No hay restaurante registrado", "Información", JOptionPane.INFORMATION_MESSAGE)  
    }  
    else if(obtenerRepartidorAleatorio()==null){ //si hay restaurante, entonces verificamos que haya repartidores para  
        JOptionPane.showMessageDialog(null, "No hay repartidores registrados", "Información", JOptionPane.INFORMATION_MESSAGE)  
    }else{ //si hay repartidores  
        setVisible(false); //ocultamos el main  
        PedidosForm pf = new PedidosForm(Main.this); //inicializamos el form de pedidos  
        pf.setVisible(true); //mostramos el frame del pedidos  
    }  
}
```

Ahora para el botón de “Registrar Pedido” tenemos el método “btnpedidoActionPerformed” el cual sigue las mismas reglas para su declaración. En cuanto a su definición seguimos la misma lógica, primero preguntamos si hay un restaurante mediante un condicional, de no haberlos se muestra el mensaje de “No hay restaurante registrado”, en caso de haber un restaurante verificamos si hay repartidores registrados mediante el método “obtenerRepartidorAleatorio()”, si el método es ==null, es decir, que no hay repartidores registrados, se muestra dicho mensaje siguiente la misma sintaxis de código del mensaje de “No hay restaurante registrado” pero cambiando solamente el mensaje que se mostrara por el de “No hay repartidores registrados. Finalmente, si el if no se cumple y el else if tampoco se cumple pasamos al código del “else” el cual se ejecuta cuando si hay restaurante y repartidores. Dentro de esta condición se inicializa el form de pedido mediante PedidosForm pf = new PedidosForm(Main.this);. Aquí es donde se crea la instancia de PedidosForm, la acción “new PedidosForm()” crea un objeto de la clase PedidosForm, cabe mencionar que la variable para la manipulación de la instancia y objeto es “pf”.

La instrucción de setVisible(false) se oculta la ventana del form Main y mediante la línea “pf.setVisible(true)” se muestra el frame del form de pedidos.

```
private void btnrepartidorActionPerformed(java.awt.event.ActionEvent evt) {
    if(r.getNombre()==null){//si no hay restaurante , mandara un mensaje para que se registre
        JOptionPane.showMessageDialog(null, "No hay restaurante registrado", "Información", JOptionPane.INFORMATION_MESSAGE);
    }
    else{//si hay restaurante
        setVisible(false); //ocultamos el main
        RepartidorForm rf=new RepartidorForm(Main.this); //mostramos el form de repartidores para registrarlos
        rf.setVisible(true); //se abre el form
    }
}
```

Para el botón de “Registrar Repartidor” del diseño del Main, tenemos el método de nombre “btnrepartidorActionPerformed” el cual como los métodos de botones anteriores sigue la misma forma de iniciar su declaración, en cuanto a la definición también sigue una lógica similar ya que primero mediante un if se pregunta si hay restaurantes registrados, de no haberlo se muestra el mensaje que ya conocemos, en caso de si haber un restaurante creamos una instancia de RepartidorForm y creamos igualmente un objeto de la misma clase, ocultamos el frame del main para después mostrar el del form de “rf” siendo esta la variable para referirse a la instancia recién creada de RepartidorForm.

```
public static void main(String args[]) {
    java.awt.EventQueue.invokeLater(new Runnable() {
        public void run() {
            new Main().setVisible(true);
        }
    });
}
```

Ahora pasamos al main del programa el cual será el punto de entrada de la aplicación y se acciona cuando el programa Java comienza a ejecutarse. El método main tiene que ser public, static, y void. Recibe un arreglo de cadenas (String[] args), que puede contener argumentos de línea de comandos.

Después en la definición del main tenemos la línea de código “java.awt.EventQueue.invokeLater(new Runnable() { })”: Esta línea utiliza la clase EventQueue para ejecutar la creación y visualización de la interfaz de usuario (UI) en el subproceso de eventos de Swing. Esto es necesario para asegurarse de que las operaciones relacionadas con la interfaz de usuario se realicen en el hilo de eventos de Swing, que es el hilo dedicado a la interfaz gráfica.

El parámetro “new Runnable()” crea una instancia anónima de la interfaz Runnable. La interfaz Runnable tiene un solo método, run(), que se ejecuta cuando el hilo es iniciado. Aquí, estamos proporcionando una implementación anónima de Runnable mediante una clase interna anónima.

La expresión java.awt.EventQueue.invokeLater(new Runnable() {...}) se utiliza en aplicaciones Swing de Java para garantizar que ciertas operaciones relacionadas con la interfaz de usuario se ejecuten en el hilo de eventos de Swing. En Java Swing, las operaciones de interfaz de usuario deben realizarse en el hilo de eventos de Swing para evitar problemas de concurrencia y asegurar la consistencia y la seguridad de la interfaz gráfica.

1.-java.awt.EventQueue.invokeLater(...): EventQueue.invokeLater es un método estático de la clase EventQueue en el paquete java.awt. Este método toma un objeto Runnable y asegura que el código dentro del método run de ese objeto se ejecute en el hilo de eventos de Swing. Este método se utiliza comúnmente en el método main de una aplicación Java Swing.

2.-new Runnable() {...}: Se crea una instancia anónima de la interfaz Runnable. La interfaz Runnable es una interfaz funcional en Java que tiene un solo método llamado run(). Al proporcionar una implementación anónima de Runnable, puedes encapsular el código que deseas ejecutar en el hilo de eventos de Swing.

En conjunto, java.awt.EventQueue.invokeLater(new Runnable() {...}) se utiliza para encapsular las operaciones de interfaz de usuario dentro del método run() de un objeto Runnable y garantizar que se ejecuten de manera segura en el hilo de eventos de Swing. Esto es importante porque la interfaz gráfica de usuario de Swing no es segura para subprocesos; por lo tanto, las operaciones de interfaz deben realizarse en el hilo de eventos de Swing para evitar problemas de concurrencia y actualización de la interfaz.

Hecho lo anterior seguimos con la línea “public void run(){}” la cual Define la implementación del método run() de la interfaz Runnable. Dentro de este método, se coloca el código que se ejecutará en el hilo de eventos de Swing.

Finalmente, dentro de la definición del método run() colocamos el código principal del programa el cual sería el Form Main por lo que mediante “new Main().setVisible(true)” Se crea una instancia de la clase Main. Luego, se llama al método setVisible(true) para hacer visible el formulario principal.

```
public Restaurante getRestaurante() {//funcion para acceder al objeto restaurante y hacer los registros de sus atributos
    return r;//regresamos el objeto de tipo restaurante
}
public List getRepartidores() {//funcion para acceder a la lista de repartidores y hacer los registros de sus atributos
    return rep;//regresamos la lista de repartidores
}
```

También se realizaron los métodos get de los atributos de la clase Main para que de esta manera tengamos acceso a dichos objetos o lista en caso de los repartidores, y podamos hacer los registros de sus atributos.

```
public String imprimir(){//funcion para imprimir repartidores
if (rep.isEmpty()) {//si la lista de repartidores esta vacia
    return "No hay repartidores registrados";//retorna esta string
}
StringBuilder resultado = new StringBuilder(); //si no, se crea un stringbuilder, que esta practicamente va a concatenar
for (Repartidor re : rep) {//ciclo foreach para que todos los repartidores se impriman
    resultado.append(re.imprimir()).append("\n");//con esta funcion haremos la gran cadena, append lo que hara es
}
return resultado.toString(); //retornamos la gran cadena
```

Para el método “imprimir” usamos la misma lógica que hemos visto en anteriores métodos similares, dicho método ahora es para imprimir a los repartidores por lo que primero se verifica que halla repartidores, de lo contrario se retorna un mensaje de “No hay repartidores registrados”, en caso de si haber repartidores, creamos una mega cadena y mediante un foreach vamos iterando sobre la lista rep en búsqueda de datos de tipo “Repartidor” usando como variable “re” para guardar los datos e igualmente mediante el operando “append()” vamos agregando los datos obtenidos del método “re.imprimir()” dentro de append a la megacadena “resultado”, para finalmente retornarla mediante “return resultado.toString();”. Mediante este método se mostrarán en el textArea a todos los repartidores registrados junto con sus atributos previamente definidos.

```
public Repartidor obtenerRepartidorAleatorio() {
    if (rep.isEmpty()) {
        return null; // si la lista está vacía, devuelve null
    }
    // crear un objeto Random para generar índices aleatorios
    Random random = new Random();
    // generar un índice aleatorio dentro del rango de la lista
    int indiceAleatorio = random.nextInt(rep.size());
    // obtener el repartidor correspondiente al índice aleatorio
    return rep.get(indiceAleatorio);
}
```

Anteriormente también vimos el método “obtenerRepartidorAleatorio()” dentro del método asignado para el botón de “Realiza Pedido” por lo que ahora lo explicaremos: Primero podemos apreciar su declaración que lleva una visibilidad public y retorna un dato de tipo “Repartidor”. Posteriormente se aprecia como tenemos una sentencia condicional if la cual pregunta si la lista “rep” esta vacía, de ser así se retorna un “null”, de lo contrario se ejecuta el resto del código el cual primero mediante la línea “Random random = new Random(,” crea una instancia de la clase Random la cual en java se utiliza para generar numero aleatorios, siendo “random” la instancia de la clase y la variable que utilizaremos para operarla.

Después en “int indiceAleatorio = random.nextInt(rep.size());” se genera un número aleatorio dentro del rango de la longitud de la lista “rep”, la parte “nextInt(rep.size())” devuelve un número entero aleatorio en

el rango [0, rep.size()]. El método “rep.size()” devuelve el tamaño de la lista. Todo esto se almacena en la variable “indiceAleatorio”.

Finalmente retornamos el elemento en la lista “rep” correspondiente al índice aleatorio generado. En otras palabras, selecciona un repartidor aleatorio de la lista de repartidores.

-rep.get(indiceAleatorio): Este método se utiliza para obtener el elemento en la posición del índice aleatorio de la lista rep. La llamada a get toma un índice como argumento y devuelve el elemento correspondiente en esa posición.

```
// Variables declaration - do not modify
private javax.swing.JButton btnmenu;
private javax.swing.JButton btnpedido;
private javax.swing.JButton btnplatillo;
private javax.swing.JButton btnrepartidor;
private javax.swing.JButton btnrestaurante;
private javax.swing.JButton btnsalir;
private javax.swing.JLabel jLabel1;
private javax.swing.JPanel jPanel1;
// End of variables declaration
}
```

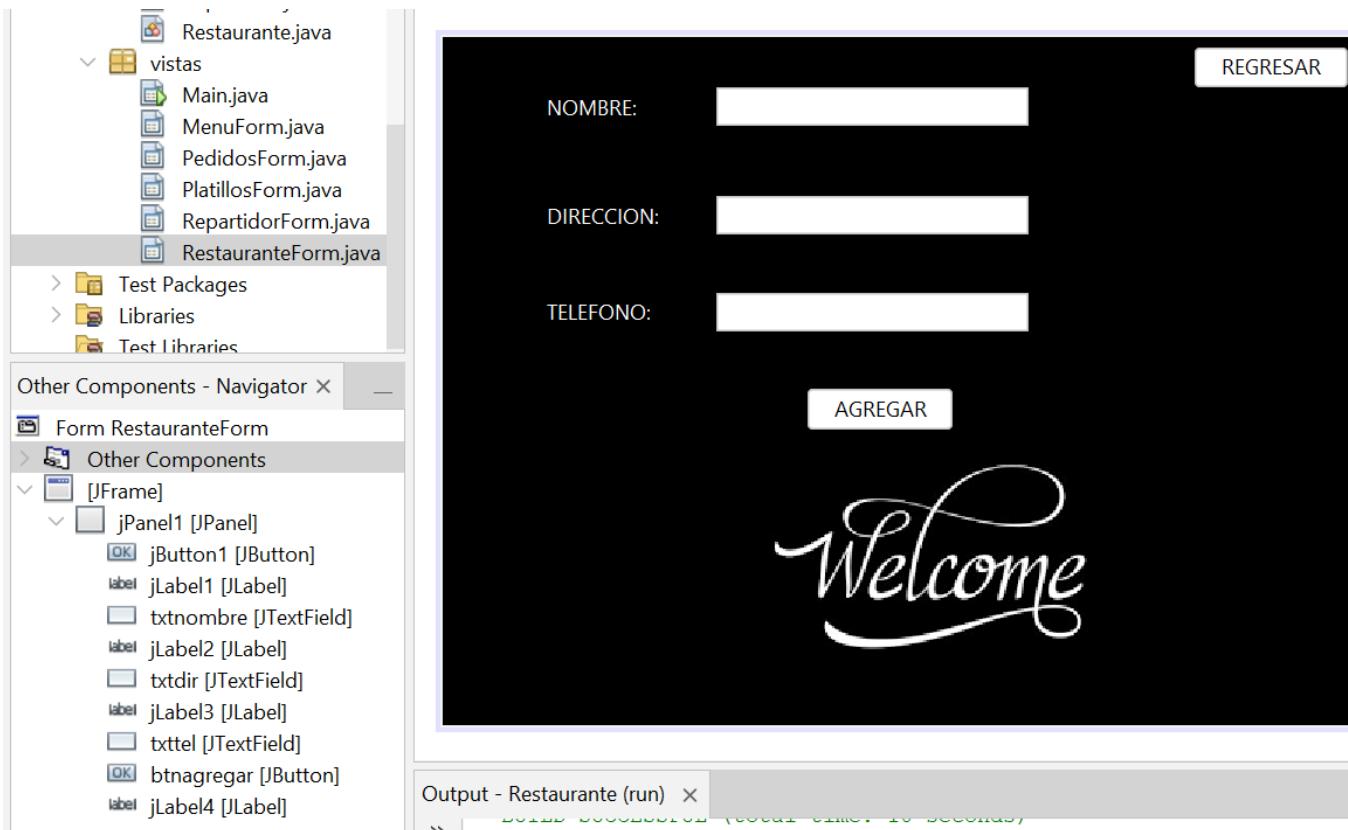
Para esta última porción del código de la clase Main debemos aclarar que son la declaración de variables de instancia para botones, etiquetas y paneles en una interfaz gráfica de usuario (GUI) utilizando la biblioteca Swing de Java. Estas variables son generadas automáticamente por el diseñador de interfaz gráfica de NetBeans (u otra herramienta similar) al diseñar visualmente la interfaz. Estas variables de instancia son generadas para proporcionar acceso a los elementos de la interfaz desde el código Java.

La notación javax.swing.JButton, javax.swing.JLabel, etc., indica los tipos de los objetos asociados a estas variables por lo que vienen seguidas del nombre que se les puso y asignó en el diseño del Form para llevar un mejor control de su uso. Estos son tipos específicos de la biblioteca Swing que representan botones, etiquetas y paneles respectivamente.

Form RestauranteForm ; RestauranteForm :: JFrame

Siguiendo la lógica del uso del programa, cuando ya se mostro el form del Main, como primera acción debemos registrar un restaurante por lo que ahora pasaremos a explicar todo lo relacionado a esta acción.

Primero observaremos la creación del diseño de la interfaz grafica para la ventana del form “RestauranteForm”.



Como podemos apreciar sigue una lógica similar a la del Main ya que solo se van seleccionando, colocando y diseñando los componentes de la ventana, pero esta vez se integra un nuevo tipo de componente el cual es una caja que sirve para registrar texto o dicho de otra forma, un objeto de tipo [JTextField]. Los demás elementos son el panel básico usado como fondo, etiquetas de texto, botones y la importación de una imagen la cual se encuentra dentro de la paquetería de “vistas”.

```
package vistas;
import javax.swing.JOptionPane;
public class RestauranteForm extends javax.swing.JFrame {
    private final Main main;//atributo el main para poder regresar
    public RestauranteForm(Main main) {
        this.main=main;//inicializamos el main
        initComponents();
        this.setLocationRelativeTo(null);//colocamos al centro el form
    }
    @SuppressWarnings("unchecked")
}
```

Generated Code

Pasando a, código del form de Restaurante poseemos observar que primero se importa el paquete de “vistas” junto con la librería Swing para utilizar cuadros de dialogo. Ya en la declaración de la clase “Restaurante” se sigue la misma estructura que vimos anteriormente por lo que se coloca el “extends javax.swing.JFrame”. Una vez dentro de la clase “RestauranteForm” comenzamos declarando un

atributo “main” de tipo “Main” para así dentro del constructor de “RestauranteForm” podamos inicializar “main” para su manipulación en el form del restaurante lo que nos permitirá poder regresar al Form Main.

Luego se usa el método initComponents(), para generar código gráfico que podamos definir y después se centra el form con “this.setLocationRelativeTo(null)”

Especificamente:

1.-public class RestauranteForm extends javax.swing.JFrame {: Esta línea declara la clase RestauranteForm que extiende la clase javax.swing.JFrame. Esto significa que RestauranteForm es una ventana de interfaz gráfica.

2.-private final Main main;: Aquí se declara un atributo main de tipo Main como final, lo que indica que este atributo no cambiará después de la inicialización. Este atributo es referencia a un objeto de la clase Main

3.-public RestauranteForm(Main main) {: Este es el constructor de la clase RestauranteForm. Toma un objeto Main como parámetro y lo asigna al atributo main. Este patrón de pasar un objeto Main al constructor sugiere que esta clase está destinada a interactuar con la instancia de Main de alguna manera.

4-.this.main = main;: Aquí se inicializa el atributo main con el valor proporcionado como parámetro al constructor. La palabra clave this se utiliza para referirse al atributo de la instancia actual.

5.-initComponents();: Parece ser una llamada a un método initComponents(). Este método probablemente se generó automáticamente y contiene la inicialización de los componentes gráficos para el formulario.

```
@SuppressWarnings("unchecked")
Generated Code

private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
    this.setVisible(false); // Oculta la ventana del apartado
    main.setVisible(true); //muestra el main
}

private void txttelActionPerformed(java.awt.event.ActionEvent evt) {

}
```

Posteriormente se codifica la supresión de ventanas de emergencia y pasamos a los métodos para los botones y elementos del diseño gráficos. Como primer método tenemos el designado para “Regresar” por lo que su declaración sigue la misma sintaxis y se orienta a eventos, ya dentro de su definición mediante “this.setVisible(false)” se oculta la ventana del “FormRestaurante” para después con “main.setVisible(true)” mostrar el form “Main” es decir la ventana principal.

Luego se observa que tenemos un método vacío generado el cual no utilizaremos pues ya hay una función que trabaja con las cajas de texto del diseño.

```

private void btnagregarActionPerformed(java.awt.event.ActionEvent evt) {
    if(txtnombre.getText().equals("")||txtdir.getText().equals("")||txttel.getText().equals("")){// si una de las cajas esta
        JOptionPane.showMessageDialog(null, "Favor de llenar todas las cajas", "Información", JOptionPane.INFORMATION_MESSAGE);
    }
    else{//si no
        String nom=txtnombre.getText(); //en el string nom guarda lo que tiene la caja de texto nombre
        String dir=txtdir.getText(); //en el string dir guarda lo que tiene la caja de texto dir
        String tel=txttel.getText(); //en el string tel guarda lo que tiene la caja de texto tel
        main.getRestaurante().setNombre(nom); //se utiliza metodos setter para agregar los valores del objeto restaurante del
        main.getRestaurante().setDireccion(dir);
        main.getRestaurante().setTelefono(tel);
        txtnombre.setText(""); //limpiar cajas
        txtdir.setText("");
        txttel.setText("");
        if(main.getRestaurante().getNombre() !=null){//si hay algo en el nombre del restaurante del main, entonces mandar un
            JOptionPane.showMessageDialog(null, "Registro exitoso", "Información", JOptionPane.INFORMATION_MESSAGE);
        }
    }
}

```

En el método generado para trabajar con el botón de “Aregar”, “btnagregarActionPerformed” tienen primero en su definición un condicional if el cual pregunta si “txtnombre.getText()” es decir la caja de texto designada para registrar el nombre, esta vacía o no contiene nada (.equals(" ")) y mediante el operador “||” es estable otra condición que de cumplirse hace que entre al código del if. Las demás condiciones que siguen del “||” son similares a la de la caja de texto, pero son para los casos de la caja para registrar la dirección y el número telefónico (txdir, txttel).

En NetBeans y en muchos entornos de desarrollo para Java que trabajan con interfaces gráficas de usuario (GUI), el método .getText() se utiliza para obtener el texto contenido en un componente de texto, como un campo de texto (JTextField).

De cumplirse cualquiera de las condiciones del if se muestra en pantalla un mensaje informativo que muestra el mensaje de “Favor de llenar todas las cajas” gracias a la línea “JOptionPane.showMessageDialog(null, “Favor de llenar todas las cajas”, “Información”, JOptionPane.INFORMATION_MESSAGE);”.

En caso contrario de estar rellenadas todas las cajas de texto se ejecuta el código dentro del “else” el cual primero guarda en el String “nom” el texto que hay dentro de “txtnombre” haciendo uso del getText(); lo que por completo se vería de la siguiente forma: “String nom=txtnombre.getText();”. Lo mismo ocurre para las demás cajas de texto guardando en “dir” lo que hay en “txdir” y en “tel” lo que hay en “txttel”.

Posteriormente accedemos a nuestra variable “main” y mediante el uso del método “getRestaurante()” asignamos a nuestro restaurante todos los datos obtenidos haciendo uso de los métodos “setNombre(nom)”, “setDireccion(dir)”, “setTelefono(tel)”.

Por ejemplo:

main.getRestaurante().setNombre(nom); Aquí se accede al objeto “Restaurante” dentro del objeto “main” y se utiliza un método setter (setNombre) para asignar el valor de “nom” al atributo “nombre” del objeto “Restaurante”. Esto asume que Restaurante tiene un método setter para el nombre el cual ya se codificó en la explicación de las clases del diagrama UML.

Finalmente vaciamos las cajas de texto mediante el método setText(“ ”) y pasamos al ultimo condicional if el cual pregunta “main.getRestaurante().getNombre() !=null” es decir si el nombre del restaurante es

diferente de null por lo tanto si hay un restaurante agregado, en termino específicos con “main.getRestaurante()” estamos accediendo al objeto “Restaurante” del objeto “main” y agregando “main.getRestaurante().getNombre()” estamos accediendo al atributo “nombre” del objeto “Restaurante” que esta en la instancia “main”.

1.-main.getRestaurante().getNombre(): esta expresión sugiere que la clase Main tiene un método llamado getRestaurante() que devuelve un objeto de tipo "Restaurante", y luego se llama al método getNombre() en ese objeto "Restaurante" para obtener el nombre del restaurante. En resumen, se está accediendo al nombre del restaurante a través de la instancia de Main que fue pasada al constructor de RestauranteForm.

De si tener un “nombre” el “Restaurante” se muestra un mensaje de “Registro exitoso” mediante la línea: “JOptionPane.showMessageDialog(null, “Registro exitoso”, “Información”, JOptionPane.INFORMATION_MESSAGE);”

```
// Variables declaration - do not modify
private javax.swing.JButton btnagregar;
private javax.swing.JButton jButton1;
private javax.swing.JLabel jLabel1;
private javax.swing.JLabel jLabel2;
private javax.swing.JLabel jLabel3;
private javax.swing.JLabel jLabel4;
private javax.swing.JPanel jPanel1;
private javax.swing.JTextField txtdir;
private javax.swing.JTextField txtnombre;
private javax.swing.JTextField txttel;
// End of variables declaration
}
```

Estas líneas son comentarios generados automáticamente por la herramienta y no afectan el comportamiento del código en sí. Aunque no alteran la lógica del programa, proporcionan información útil sobre la estructura de la interfaz gráfica.

Estas líneas en particular están relacionadas con la declaración de variables para los elementos gráficos (botones, etiquetas, paneles y campos de texto) en un formulario.

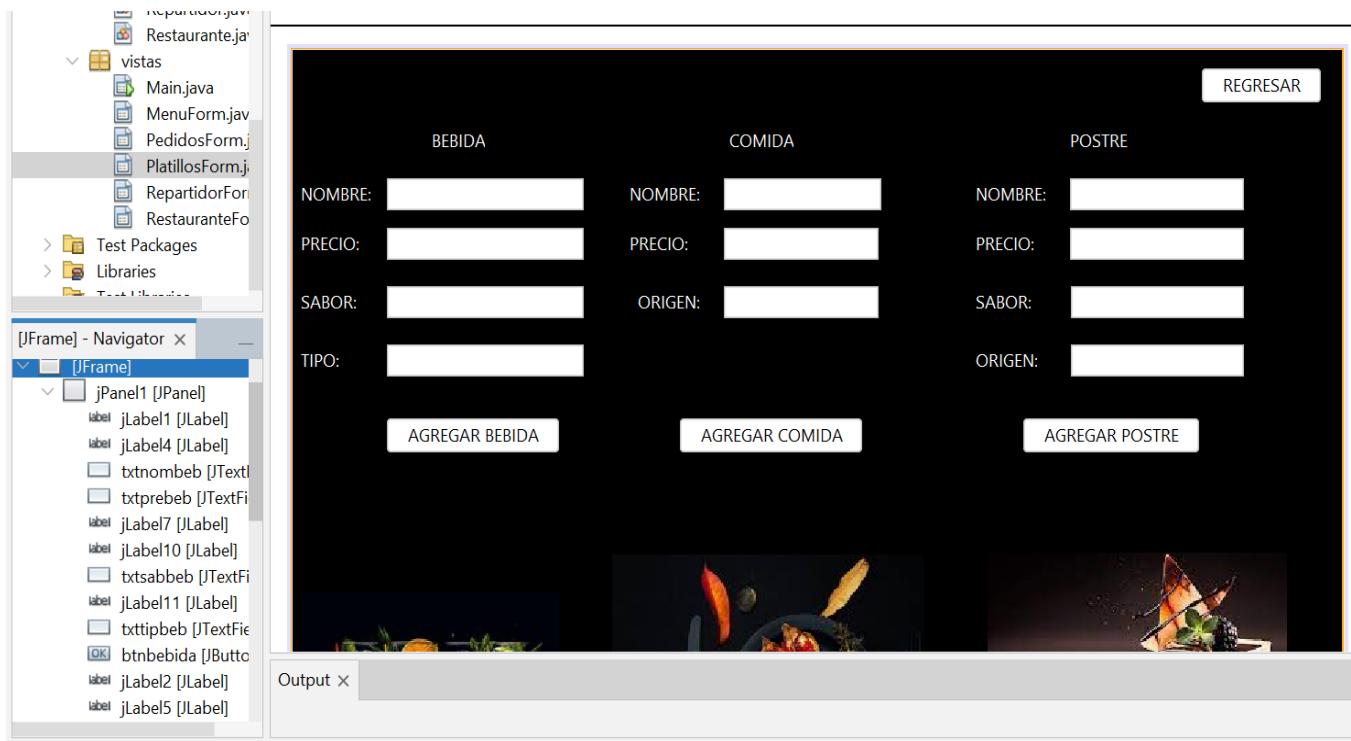
Por ejemplo:

private javax.swing.JButton btnagregar; Declarar una variable para un botón o variable llamado "btnagregar" de tipo "JButton".

Form PlatillosForm ; PlatillosForm :: JFrame

Ya estamos llegando a las ultimas partes del programa las cuales siguen una lógica similar a anteriores clases y forms explicados con anterioridad por lo que ahora solo detallaremos las cosas nuevas que se vean en el código.

Primero tenemos que ver el diseño del PlatilloForm:



Como se observa tenemos elementos ya antes usados como botones, cajas de texto, paneles y etiquetas por lo que no es necesario profundizar en la explicación.

```

package vistas;
import modelo.Bebida;
import modelo.Comida;
import modelo.Postre;
import javax.swing.JOptionPane;

public class PlatillosForm extends javax.swing.JFrame {
    private final Main main; //mandamos el main como atributo para poder regresar

    public PlatillosForm(Main main) {
        this.main=main; //inicializamos el form
        initComponents();
        this.setLocationRelativeTo(null); //colocamos al centro
    }
    @SuppressWarnings("unchecked")
    Generated Code

    private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
        this.setVisible(false); // Oculta la ventana del apartado
        main.setVisible(true); //muestra el main
    }
}

```

Pasando al código del “PlatillosForm” se tiene primero la declaración del paquete “vistas” para el uso de las imágenes dentro de dicho paquete, luego tenemos la importación de las clases “Comida”, “Bebida” y “Postre” que están dentro del paquete “modelo”, finalmente tenemos la “import

`javax.swing.JOptionPane;`" esta librería en Java se utiliza para hacer accesible la clase JOptionPane del paquete javax.swing. JOptionPane es una clase que proporciona métodos para mostrar fácilmente cuadros de diálogo de mensajes, entrada y confirmación en aplicaciones de interfaz gráfica de usuario (GUI) en Java.

Ya estando en la declaración de la clase "PlatillosForm" seguimos la misma sintaxis extendiendo "javax.swing.JFrame". Una vez en la definición hacemos lo mismo que se hizo en "RestauranteForm" por lo que mandamos el "Main" como atributo en una variable final de nombre "main", iniciamos el constructor de "PlatillosForm" pasando como atributo "main" para inicializarlo dentro de dicho constructor, luego usamos el método para generar el código de la interfaz grafica y centramos la ventana grafica. Cabe mencionar que también se coloca la línea de código para suprimir los mensajes de advertencia.

Como primer método tenemos el designado para el botón de la interfaz gráfica "Regresar" el cual sigue la misma lógica y sintaxis que se utilizo en la clase "RestauranteForm".

```
private void txtnombebActionPerformed(java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
}  
  
private void txtnomposActionPerformed(java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
}  
  
private void txtorgposActionPerformed(java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
}
```

Tenemos como parte del código generado los métodos mostrados en la imagen sin embargo no se utilizarán ya que manipularemos todas las acciones deseadas con los métodos que se mostrarán a continuación:

```
private void btnbebidaActionPerformed(java.awt.event.ActionEvent evt) {  
    if(txtnombeb.getText().equals("") || txtprebeb.getText().equals("") || txtsabbeb.getText().equals("") || txttipbeb.getText().equals(""));  
        JOptionPane.showMessageDialog(null, "Favor de rellenar todas las cajas", "Información", JOptionPane.INFORMATION_MESSAGE);  
    else{  
        String nom=txtnombeb.getText(); //lo que hay en nombre lo guardara en la cadena nombre  
        int precio=Integer.parseInt(txtprebeb.getText()); //lo que hay en precio lo guardara en el int de precio haciendo un  
        String sab=txtsabbeb.getText(); //lo que hay en sabor lo guardara en la cadena sab  
        String tip=txttipbeb.getText(); //lo que hay en tipo lo guardara en la cadena tip  
        Bebida b=new Bebida(nom,precio,tip,sab);  
        main.getRestaurante().agregarAlMenu(b); // accedemos al main y a su objeto de tipo restaurante y de ahí accedemos al  
        txtnombeb.setText(""); // limpiamos las cajas  
        txtprebeb.setText("");  
        txtsabbeb.setText("");  
        txttipbeb.setText("");  
        JOptionPane.showMessageDialog(null, "Registro Exitoso", "Información", JOptionPane.INFORMATION_MESSAGE); //mandamos un mensaje de exito  
    }  
}
```

Este método “btnbebidaActionPerformed” es el designado para el botón “Aregar Bebida” y sigue exactamente la misma lógica del método “btntagregarActionPerformed” designado para el botón “Aregar” del form del restaurante. Lo único que cambia son el nombre de las cajas de texto designadas en el diseño para el registro de datos junto con el nombre de las variables para guardar dicha información. Los métodos para obtener y asignar la información son los mismos, al igual que las líneas de código para mostrar mensajes informativos, cabe mencionar que también se sigue la misma secuencia lógica del condicional if por lo que se verifica primero que por ejemplo: al agregar una bebida todas las cajas de texto respectivas deben estar llenas, de no ser así se muestra su respectivo mensaje informativo, en caso de estar completas todos los cuadros de texto se ejecuta el código del else que obtiene, asigna y guarda la información en sus respectivas variables, limpia el texto y muestra el mensaje de “Registro Exitoso”.

Lo único diferente que tenemos son tres líneas de código nuevas las cuales explicaremos.

1.- int precio=Integer.parseInt(txtprebeb.getText()); :

La línea de código int precio = Integer.parseInt(txtprebeb.getText()); se utiliza para convertir el texto ingresado en un campo de texto (JTextField) llamado txtprebeb a un valor entero (int).

txtprebeb.getText(): txtprebeb es un objeto de la clase JTextField que representa un campo de texto en una interfaz gráfica de usuario (GUI). getText() es un método de la clase JTextField que devuelve el texto ingresado por el usuario en ese campo.

Integer.parseInt(...): Integer.parseInt es un método estático de la clase Integer en Java que toma una cadena de texto como argumento y devuelve el valor entero correspondiente. En este caso, se está utilizando para convertir el texto obtenido de txtprebeb.getText() a un entero.

int precio = ...: Se declara una variable precio de tipo int y se le asigna el valor obtenido después de convertir el texto a entero.

En resumen, esta línea de código se utiliza para obtener el valor numérico ingresado por el usuario en el campo de texto txtprebeb y almacenarlo en la variable precio como un número entero.

Dato: Es importante mencionar que esta operación puede lanzar una excepción NumberFormatException si el texto ingresado no es un número válido, por lo que es posible que se deba manejar esta excepción para garantizar la robustez del programa.

2.- Bebida b=new Bebida(nom,precio,tip,sab); :

Esta línea está creando una nueva instancia del objeto “Bebida” y asignándola a la variable “b” con los parámetros de “nom”, “precio”, “tip” y “sab” las cuales son las variables que tienen guardados los datos que se registraron previamente con respecto a su sección como el tipo, sabor, precio o el nombre. Dicho de otra forma, se está instanciando un objeto de nombre “b” de tipo “Bebida”. Esto nos sirve para la siguiente línea de código.

3-.main.getRestaurante().agregarAlMenu(b);

Esta línea accede al objeto “Restaurante” del atributo “main” que en realidad es el nombre de la variable designado para manipular la clase “Main”, para posteriormente mediante el método “agregarAlmenu()” pasemos como parámetro el nombre del objeto “b” que contiene todos los datos registrados previamente de una “Bebida” para agregarlos al “MenuForm”.

main.getRestaurante().agregarAlMenu(b); sugiere que estás utilizando un objeto main que tiene un método llamado getRestaurante() que a su vez devuelve un objeto de tipo Restaurante. Luego, estás llamando al método agregarAlMenu(b) en ese objeto Restaurante, donde b es una instancia de la clase Bebida por lo que estamos pasando la bebida b como argumento al método.

Ya entendidas las nuevas tres líneas de código es necesario que aclaremos que también están los métodos “btncomidaActionPerformed” para el botón “Agregar Comida” y “btnpostreActionPerformed” para el botón “Agrega Postre” los cuales sigue la misma lógica de programación del método creado para el botón de “Agregar Bebida” siendo obviamente lo único que cambia el nombre de las cajas de texto designadas al apartado, el nombre de las variable para el guardado de la información, la creación del objeto según el tipo del platillo con sus respectivos parámetros y el uso del método para agregar al menu el objeto creado para el platillo que se registró.

```
private void btncomidaActionPerformed(java.awt.event.ActionEvent evt) {  
    if(txtnomcom.getText().equals("") || txtprecom.getText().equals("") || txtorgcom.getText().equals("")){//si las cajas de co  
        JOptionPane.showMessageDialog(null, "Favor de rellenar todas las cajas", "Información", JOptionPane.INFORMATION_MESSAGE);  
    }else{  
        String nom=txtnomcom.getText(); //lo que hay en nombre lo guardara en la cadena nombre  
        int precio=Integer.parseInt(txtprecom.getText()); //lo que hay en precio lo guardara en el int de precio haciendo una  
        String org=txtorgcom.getText(); // lo que hay en org lo gusradar en la cadena org  
        Comida c=new Comida(nom,precio,org);  
        main.getRestaurante().agregarAlMenu(c); // accedemos al main y a su objeto de tipo restaurante y de ahí accedemos al  
        txtnomcom.setText(""); // limpiamos las cajas  
        txtprecom.setText("");  
        txtorgcom.setText("");  
        JOptionPane.showMessageDialog(null, "Registro Exitoso", "Información", JOptionPane.INFORMATION_MESSAGE); //mandamos un  
    }  
}
```

```

private void btnpostreActionPerformed(java.awt.event.ActionEvent evt) {
    if(txtnompos.getText().equals("")||txtprepos.getText().equals("")||txtsabpos.getText().equals("")||txtorgpos.getText().equals(""))
        JOptionPane.showMessageDialog(null, "Favor de rellenar todas las cajas", "Información", JOptionPane.INFORMATION_MESSAGE);
    else{
        String nom=txtnompos.getText(); //lo que hay en nombre lo guardara en la cadena nombre
        int precio=Integer.parseInt(txtprepos.getText()); //lo que hay en precio lo guardara en el int de precio haciendo un
        String sab=txtsabpos.getText(); //lo que hay en saboe lo guardara en la cadena sab
        String org = txtorgpos.getText(); //lo que hay en origen lo guardara en la cadena org
        Postre p=new Postre(nom,precio,sab,org);
        main.getRestaurante().agregarAlMenu(p); // accedemos al main y a su objeto de tipo restaurante y de ahí accedemos al
        txtnompos.setText(""); // limpiamos las cajas
        txtprepos.setText("");
        txtsabpos.setText("");
        txtorgpos.setText("");
        JOptionPane.showMessageDialog(null, "Registro Exitoso", "Información", JOptionPane.INFORMATION_MESSAGE); //mandamos
    }
}

```

Finalmente, como en todos los Forms tenemos la sección de código generado automáticamente y destinada para el nombre de nuestras variables de la interfaz gráfica o declaración de nuestras variables de instancia para los elementos gráficos.

```

// Variables declaration - do not modify
private javax.swing.JButton btnbebida;
private javax.swing.JButton btncomida;
private javax.swing.JButton btnpostre;
private javax.swing.JButton jButton1;
private javax.swing.JLabel jLabel1;
private javax.swing.JLabel jLabel10;
private javax.swing.JLabel jLabel11;
private javax.swing.JLabel jLabel12;
private javax.swing.JLabel jLabel13;
private javax.swing.JLabel jLabel14;
private javax.swing.JLabel jLabel15;
private javax.swing.JLabel jLabel16;
private javax.swing.JLabel jLabel17;
private javax.swing.JLabel jLabel2;
private javax.swing.JLabel jLabel3;
private javax.swing.JLabel jLabel4;
private javax.swing.JLabel jLabel5;
private javax.swing.JLabel jLabel6;
private javax.swing.JLabel jLabel7;
private javax.swing.JLabel jLabel8;
private javax.swing.JLabel jLabel9;
private javax.swing.JPanel jPanel1;
private javax.swing.JTextField txtnombeb;
private javax.swing.JTextField txtnomcom;

```

```

private javax.swing.JPanel jPanel1;
private javax.swing.JTextField txtnombeb;
private javax.swing.JTextField txtnomcom;
private javax.swing.JTextField txtnompos;
private javax.swing.JTextField txtorgcom;
private javax.swing.JTextField txtorgpos;
private javax.swing.JTextField txtprebeb;
private javax.swing.JTextField txtprecom;
private javax.swing.JTextField txtprepos;
private javax.swing.JTextField txtsabbeb;
private javax.swing.JTextField txtsabpos;
private javax.swing.JTextField txttipbeb;
// End of variables declaration

```

Form MenuForm ; MenuForm :: JFrame

Ya hecho el registro de un restaurante y algunos platillos, podemos acceder al botón de “Mostrar Menu” que esta en el form “Main” por lo que lo explicaremos su funcionamiento.



Para el apartado del diseño del “MenuForm” tenemos la imagen que se muestra , la cual solo integra como nuevo elemento un Panel Scroll el cual nos integra la opción de una barra para poder desplazarnos

en la aplicación y observar completamente los datos registrados, en dicho panel tenemos su respectiva área de texto que es donde se mostraran toda la información obtenida previamente.

```
package vistas;

public class MenuForm extends javax.swing.JFrame {
    private final Main main;//atributo el main para poder regresar

    public MenuForm(Main main) {
        this.main=main;//inicializamos el main
        initComponents();
        this.setLocationRelativeTo(null);//colocamos al centro el form
        jLabel1.setText("BIENVENIDO A "+main.getRestaurante().getNombre());//en el inicio del form, colocamos un label que muestra el nombre del restaurante
        jLabel2.setText("DIRECCION: "+main.getRestaurante().getDireccion());//en el inicio del form, colocamos un label que muestra la dirección del restaurante
        jLabel3.setText("TELEFONO: "+main.getRestaurante().getTelefono());//en el inicio del form, colocamos un label que muestra el teléfono del restaurante
        jTextArea1.setEditable(false);//hacemos que el textArea no sea editable, esto porque es solo para informar
        jTextArea1.append(main.getRestaurante().imprimir());//en la textArea, mandamos a poner a los objetos del menu, esto es para que se imprima todo el menu
    }

    @SuppressWarnings("unchecked")
    // Generated Code
```

Ya en la parte del código tenemos primero la declaración del paquete “vistas” , después tenemos la declaración de la clase “MenuForm” con la lógica grafica que se ha estado utilizando. Dentro de la clase primero se declara un atributo final “main” de tipo “Main” para manipular la clase “Main” desde el form en el que nos encontramos.

Después pasamos a la definición del constructor de la clase, pasando como parámetro “main” y dentro de la definición se inicializa el main, se llama al método generador de código grafico y se coloca la ventana grafica en el centro. También tenemos nuevas acciones que asignan texto a nuestras etiquetas en el diseño de la interfaz por lo que, por ejemplo: la etiqueta “jLabel1” se destina para la muestra del nombre del restaurante por lo que mediante el método “setText()” logramos el cometido, quedando finalmente la línea de código.

```
"jLabel1.setText("BIENVENIDO A "+main.getRestaurante().getNombre());"
```

Cabe destacar que set asigna el texto a la etiqueta jLabel1 y dentro del método set se coloca el texto que se agregara por lo que gracias a la parte “main.getRestaurante().getNombre()” estamos obteniendo el nombre del restaurante mediante el método que se declaró en la clase “Main” haciendo uso de su variable “main”.

El mismo proceso ocurre para las demás etiquetas cambiando solamente el nombre de la etiqueta a la que se le asignara su respectivo texto y el ultimo método get que obtiene el atributo correspondiente a la etiqueta, ya sea “getDireccion()” o “getTelefono”.

Cabe destacar que mediante “jTextArea1.setEditable(false)” estamos diciendo que el textArea no puede editarse ya que solo esta para fines informativos.

Con “jTextArea1.append(main.getRestaurante().imprimir());” estamos diciendo que, en la textArea, mandamos a poner a los objetos del menu, esto accediendo al atributo main, después al objeto restaurante y de ahí al método imprimir que imprimirá todo el menu

```

@SuppressWarnings("unchecked")
Generated Code

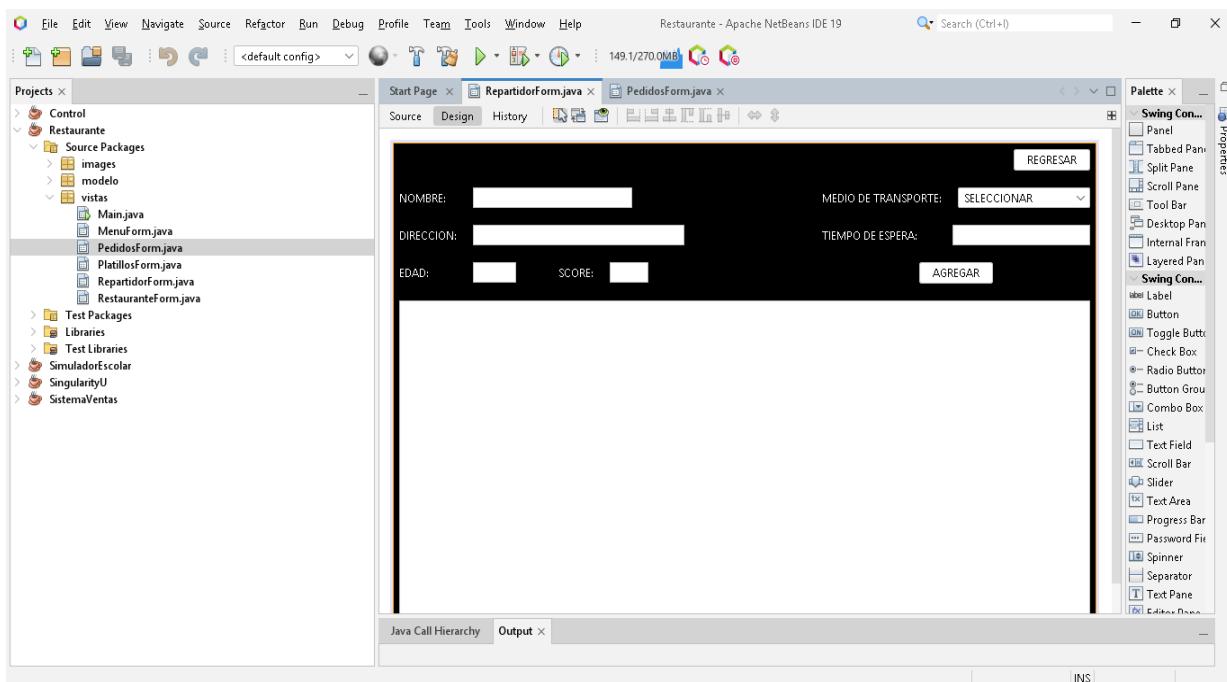
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
    this.setVisible(false); // Oculta la ventana del apartado
    main.setVisible(true); //muestra el main
}

// Variables declaration - do not modify
private javax.swing.JButton jButton1;
private javax.swing.JLabel jLabel1;
private javax.swing.JLabel jLabel2;
private javax.swing.JLabel jLabel3;
private javax.swing.JLabel jLabel4;
private javax.swing.JPanel jPanel1;
private javax.swing.JScrollPane jScrollPane1;
private javax.swing.JTextArea jTextArea1;
// End of variables declaration
}

```

Finalmente tenemos la supresión de mensajes de advertencia, la definición del método para el botón “Regresar” y la declaración de nuestras variables de la interfaz gráfica generadas automáticamente.

Form RepartidorForm ; RepartidorForm :: JFrame



```

package vistas;
import javax.swing.JOptionPane;
import modelo.Repartidor;
public class RepartidorForm extends javax.swing.JFrame {
    private final Main main; //mandamos el main como atributo para poder regresar
    public RepartidorForm(Main main) {
        this.main=main; //inicializamos el form
        initComponents();
        this.setLocationRelativeTo(null); //colocamos al centro
        jTextArea.setEditable(false); //la textarea no sera editable
        jTextArea.append(str:main.imprimir()); //la text area recibira la gran cadena del main, de la lista
    }
}

```

En este from se importa la clase repartidor, esto para que se construyan los objetos de esta clase y se agreguen a la lista de objetos de tipo repartidor que tenemos en el main, esto para llevar un registro y seguimiento de los repartidores, se tiene como atributo el main, esto hace que podamos acceder a el sin problema así como regresar al form, se coloca al centro el form con la línea de locationRelativeTo y el jTextArea se inicia sin modificación y se accede al main para acceder al método imprimir , que este regresa una cadena en donde se tienen todos los repartidores registrados, esto hace que al iniciar el form se impriman los datos en el jTextArea.

```

private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
    this.setVisible(false); // Oculta la ventana del apartado
    main.setVisible(true); //muestra el main
}

```

En esta parte es el botón de regresar, este hará que al presionarse se oculte el form y aparezca el main.

```

private void btnagregarActionPerformed(java.awt.event.ActionEvent evt) {
    agregar(); //metodo para agregar los repartidores
    jTextArea1.setText(""); //limpiamos el jarea
    jTextArea1.append(str:main.imprimir()); //volvemos a imprimir para saber si se adirio el repartidor
}

```

El botón agregar lo que hace es mandar a llamar el método agregar y la JTextArea se limpia y se vuelve a imprimir, esto como forma de actualización.

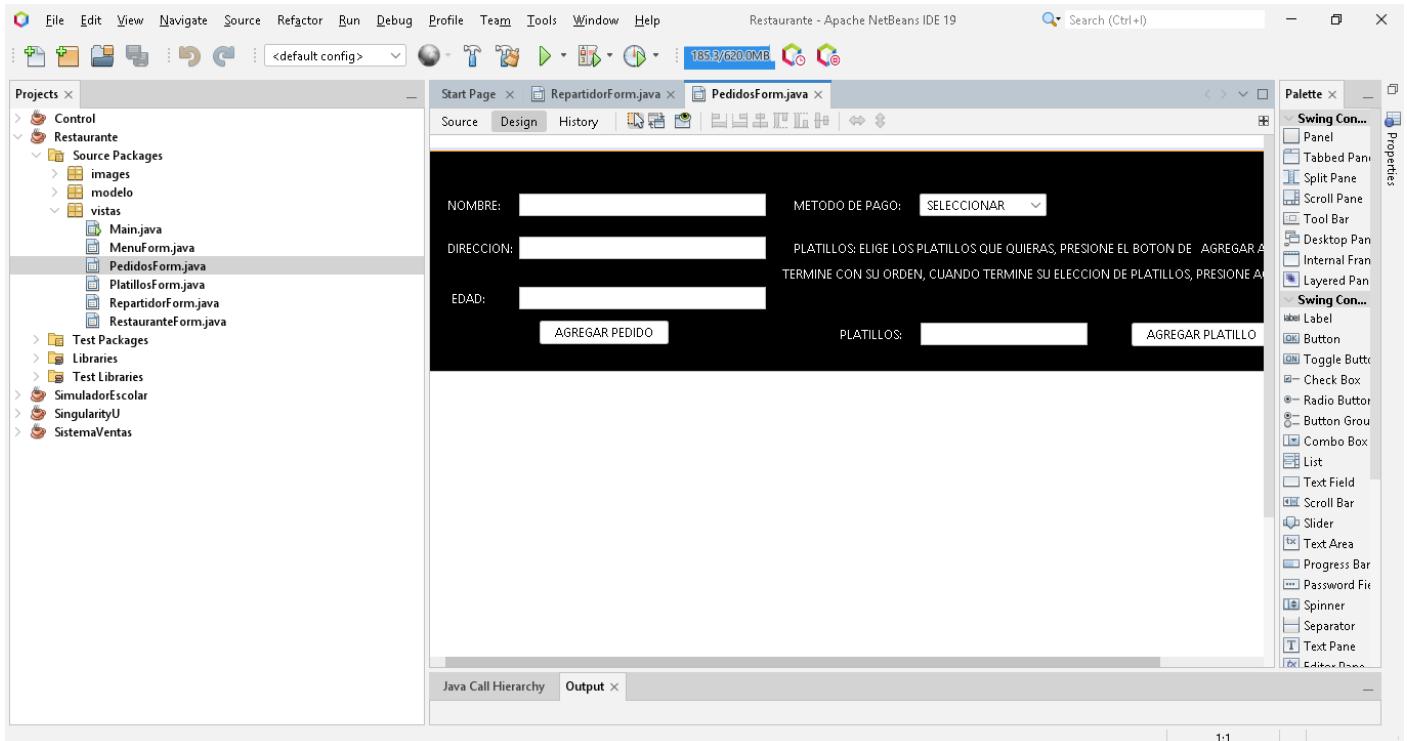
```

public void agregar() {
    if(txtnom.getText().equals(anObject: "") || txtdir.getText().equals(anObject: "") || txted.getText().equals(anObject: ""))
        JOptionPane.showMessageDialog(parentComponent: null, message: "Favor de rellenar todas las cajas", title: "")
    else{
        String nom=txtnom.getText(); //en la string nom se guarda lo que hay en la caja nom
        String dir=txtdir.getText(); //en la string dir se guarda lo que hay en la caja dir
        int ed=Integer.parseInt(txted.getText()); //en el int ed se guarda lo que hay en la caja ed hacie
        String med=cmbmed.getSelectedItem().toString(); //en la string med se guarda lo que hay en la comb
        String tiem=txttiem.getText(); //en el string tiem se guarda lo que hay en la cja tiem
        double score=Double.parseDouble(txtscore.getText()); //en el double score se guarda lo que hay en
        Repartidor nuevorep=new Repartidor(transporte: med, tiempo: tiem, score, nombre: nom, direccion: dir, edad: ed); //
        main.agregarRepartidor(r: nuevorep); //accedemos al main a la lista de repartidores y agregamos un re
        txtnom.setText(""); //limpiamos cajas
        txtdir.setText("");
        txted.setText("");
        txttiem.setText("");
        txtscore.setText("");
        JOptionPane.showMessageDialog(parentComponent: null, message: "Registro Exitoso", title: "Información", messa
    }
}

```

El método agregar al inicio si las cajas están vacías mandan un mensaje de error, si no, lo que hay en las cajas se guardara en string e int correspondientes a los tipos de datos de los atributos de la clase repartidor, posteriormente se crea un objeto de tipo repartidor y se llama el constructor con los atributos correspondientes, luego se manda a llamar al main y a su vez al método agregar repartidor con el repartidor recién construido, esto hará que en el main tenga un objeto de tipo repartidor agregado a la lista general de repartidores del main, luego se limpian las cajas y se muestra un mensaje de registro exitoso

JFRAMEFORM PEDIDOSFORM;PEDIDOS FROM :: JFrame



```

package vistas;
import javax.swing.JOptionPane;
import modelo.Repartidor;
import modelo.Receta;
import modelo.Pedido;
import java.util.List;
import java.util.ArrayList;
public class PedidosForm extends javax.swing.JFrame {
    private final Main main;//mandamos el main como atributo para poder regresar
    int total=0;//hacemos un atributo total y se incializa en 0
    List<String> pla;//arreglo de platillos, de tipo string
    public PedidosForm(Main main) {
        this.main=main;//inicializamos el main
        initComponents();
        pla = new ArrayList<>();//inicializamos la lista
        this.setLocationRelativeTo(null);//colocamos al centro
        jTextArea1.setEditable(false);//hacemos que el textarea no sea editable
        jTextArea1.append(main.getRestaurante().imprimirPedido());//accedemos al objeto restaurante del main
    }
}

```

El form inicia con la importación de la clase receta y pedido, el main es atributo de tipo final, esto para poder acceder a sus datos y poder ingresar y regresar entre los 2 form, hay un atributo llamado total, hará que se haga el conteo del dinero que se debe de pagar, y una lista de string de platillos, esto guardara los platillos seleccionados, en el constructor se inicializa el main el arreglo de platillos, se coloca

al centro el form y el JTextArea se vuelve no editable además de que imprimirá los datos del objeto restaurante hecho en el main y luego de obtener ese objeto, se manda a llamar al método dentro de restaurante llamado imprimir pedido, que es una string y se imprime en el JTextArea.

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
    this.setVisible(b: false); // Oculta la ventana del apartado
    main.setVisible(b: true); // muestra el main
}
```

El botón de regresar hará que este form se quite y el main aparezca.

```
private void btnplatilloActionPerformed(java.awt.event.ActionEvent evt) {
    agregarPlatillo(); //funcion agregar platillo
    limpiarPlatillo(); //limpiamos caja
}
```

En el botón agregar platillo, se manda a llamar 2 métodos, uno de agregación y otro limpia la caja

```
private void btnpedidoActionPerformed(java.awt.event.ActionEvent evt) {
    agregarPedido(); //funcion para agregar pedido
    limpia(); //limpiamos cajas
    jTextArea1.setText(t: ""); //limpiamos text area
    jTextArea1.append(str:main.getRestaurante().imprimirPedido()); //imprimimos
}
...
```

En el botón de agregar pedido se mandan a llamar 2 métodos, uno agrega el pedido y el otro limpia las cajas, así como se limpia el JTextArea y se vuelve a imprimir los pedidos, esto como método de actualización

```
public void agregarPedido() {
    if(txtnom.getText().equals(anObject: "") || txtdir.getText().equals(anObject: "") || txtedad.getText().equals(anObject: ""))
        JOptionPane.showMessageDialog(parentComponent: null, message: "Favor de Rellenar todas las cajas o haber")
    else //si no
        String nom=txtnom.getText(); //lo que hay en la caja de nombre, guardar en el string nom
        String dir=txtdir.getText(); //lo que hay en la caja de dirección, guardar en el string dir
        int edad=Integer.parseInt(s: txtedad.getText()); //lo que hay en la caja de edad, guardar en el int
        String pago=cmbpago.getSelectedItem().toString(); //lo que hay en la caja de pago, guardar en el string
        Repartidor r=main.obtenerRepartidorAleatorio(); //obtenemos un repartidor que se guardara como r y
        Pedido p = new Pedido(new ArrayList<>(c: pla), nombre: nom, dirección: dir, edad, r, metpago: pago, total); //llamar
        main.getRestaurante().agregarPedido(p); //llamamos al main, para obtener el restaurante que es el objeto
        total=0; //el total vuelve a 0
        pla.clear(); //limpiamos la lista de platillos
        JOptionPane.showMessageDialog(parentComponent: null, message: "Pedido agregado", title: "Información", messageType: JOptionPane.INFORMATION_MESSAGE)
    }
}
```

En el método de agregar pedido se hace una confirmación en donde se ve si las cajas de texto están vacías, si el arreglo de string que guarda los platillos está vacío, si es así, cualquiera de estas, mandara un mensaje de error, si no, lo que hay en las cajas se guardaran en variables correspondientes para la agregación en el constructor de pedido, se creara una variable de tipo repartidor y se le asignara uno que contiene el main y que regresa un repartidor aleatorio y además se agregara un nuevo arreglo de tipo platillo, y el total, esto para evitar que se reescriba la lista de platillos, se agregara el total y el platillo

de los atributos de ese form y el total se regresara a 0, y el arreglo de platillos se limpiara, esto para que no haya reescrituras en ninguno de los objetos agregados, además de que se accederá al main y de ahí obtenemos su restaurante, llamamos el método de agregar pedidos y el agregamos el pedido recién construido, al final se agrega un mensaje de confirmación.

```
public void agregarPlatillo(){
    if(txtplatillo.getText().equals("")){
        JOptionPane.showMessageDialog(null, "Agrega algo a la caja", "Información", JOptionPane.INFORMATION_MESSAGE);
    }else{
        String platillo=txtplatillo.getText();
        Receta r=main.getRestaurante().buscarPlatillo(leccion:platillo);
        if(r==null){
            JOptionPane.showMessageDialog(null, "El platillo o bebida no existe", "Error", JOptionPane.ERROR_MESSAGE);
        }else{
            pla.add(platillo);
            total+=r.getPrecio();
            JOptionPane.showMessageDialog(null, "El platillo o bebida agregada", "Información", JOptionPane.INFORMATION_MESSAGE);
        }
    }
}
```

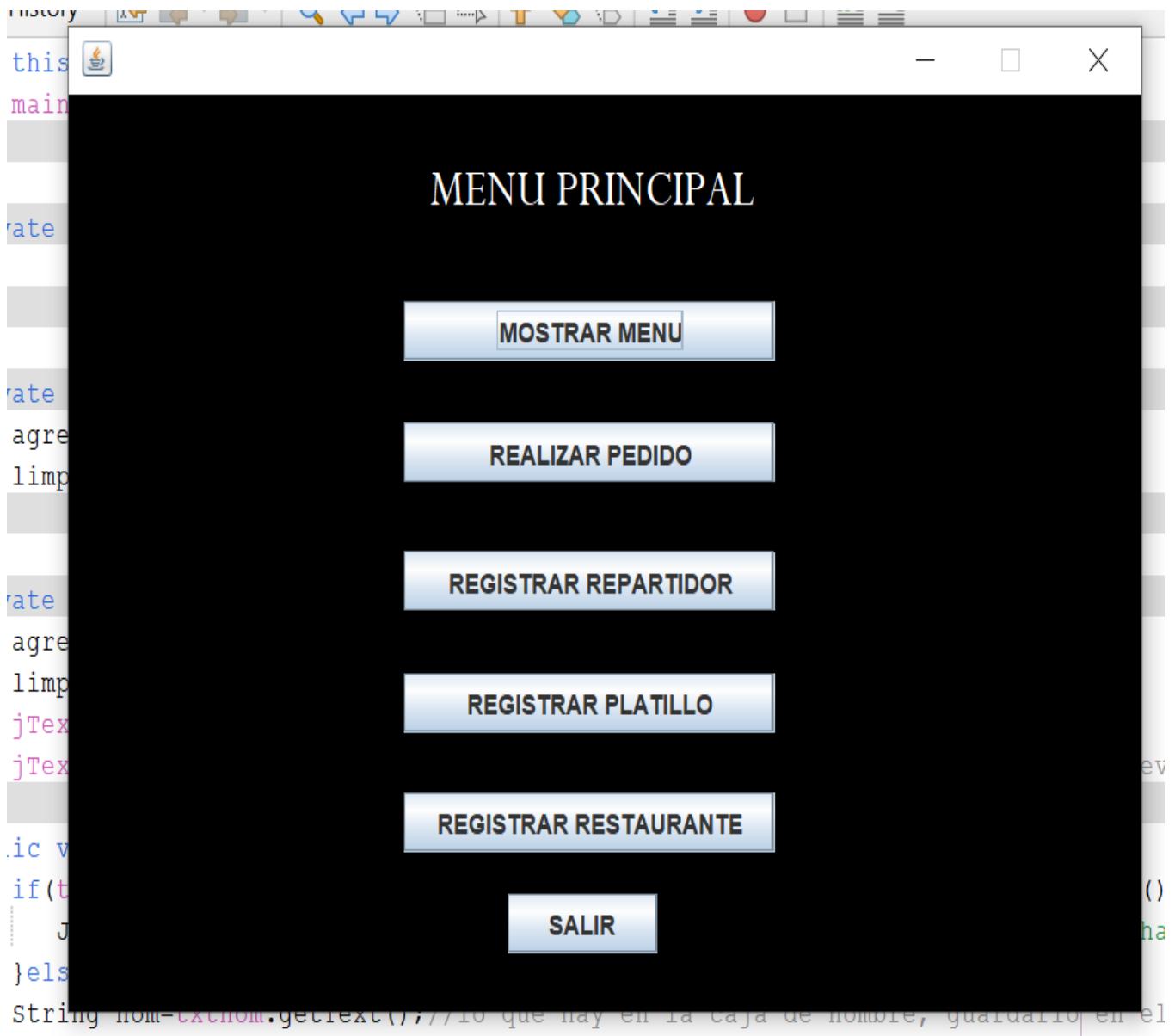
En el método agregar platillo, se confirma si la caja está vacía, si es así , pondrá un mensaje de error, si no, lo que hay en platillo lo guardara en una variable String, luego creamos e iniciamos un objeto de tipo receta el cual se le asignara un objeto de tipo receta, este objeto será uno accediendo al main, luego al objeto restaurante del main, y este tiene los platillos, por lo que se usara el método buscar platillo, que buscara la coincidencia con el nombre del platillo puesto en el main y retorna un objeto si se encuentra o un null en caso de que no exista, si es verdad de que es null, se mostrara un mensaje de no existencia, si no, en la lista de platillos de este form se agregara el nombre del platillo, el total de incrementar con el número de precio que hay en el objeto receta y se mostrara un mensaje de confirmación.

```
public void limpiarPlatillo(){
    txtplatillo.setText("");
}
public void limpia(){
    txtnom.setText("");
    txtdir.setText("");
    txtedad.setText("");
}
```

Métodos usados para la limpieza de cajas de texto.

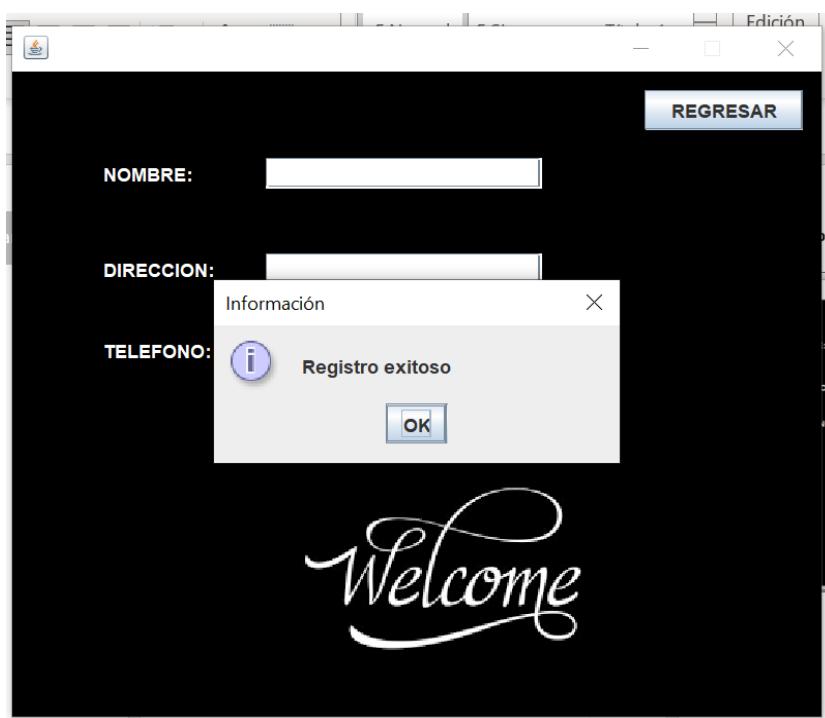
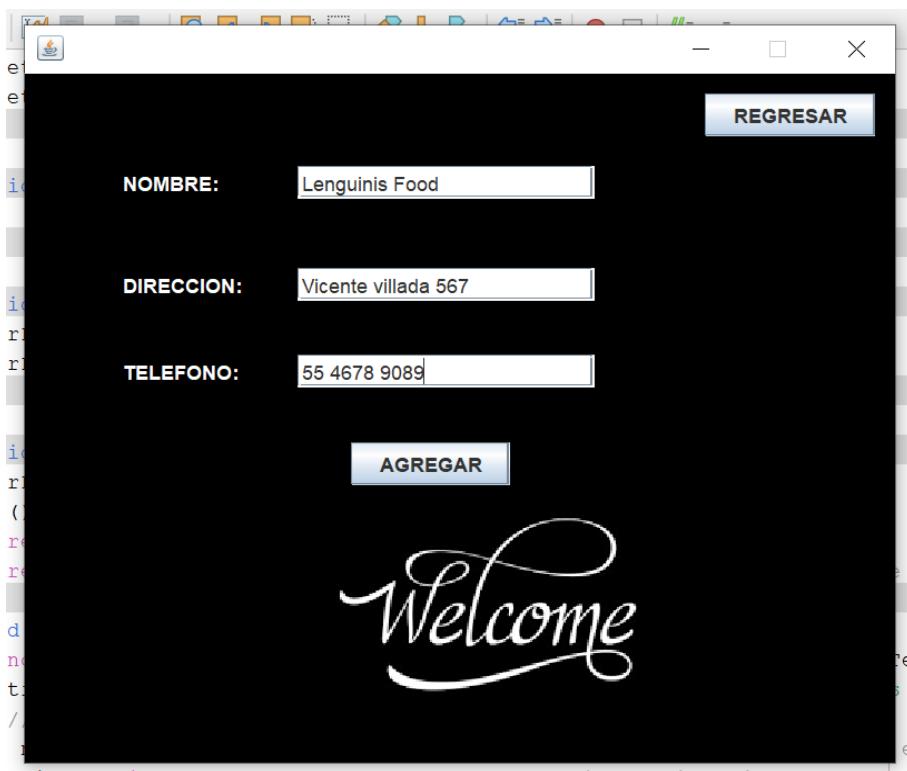
Funcionamiento

Prueba de ejecución, resultados de salida y/o capturas de pantalla



Menu principal, se selecciona el botón “REGISTRAR RESTAURANTE”

Se ingresan los datos del restaurante y se selecciona el botón “AGREGAR”



Hacemos clic en “REGRESAR” una vez registrado el restaurante y de vuelta al menu principal vamos a “REGISTRAR PLATILLO”

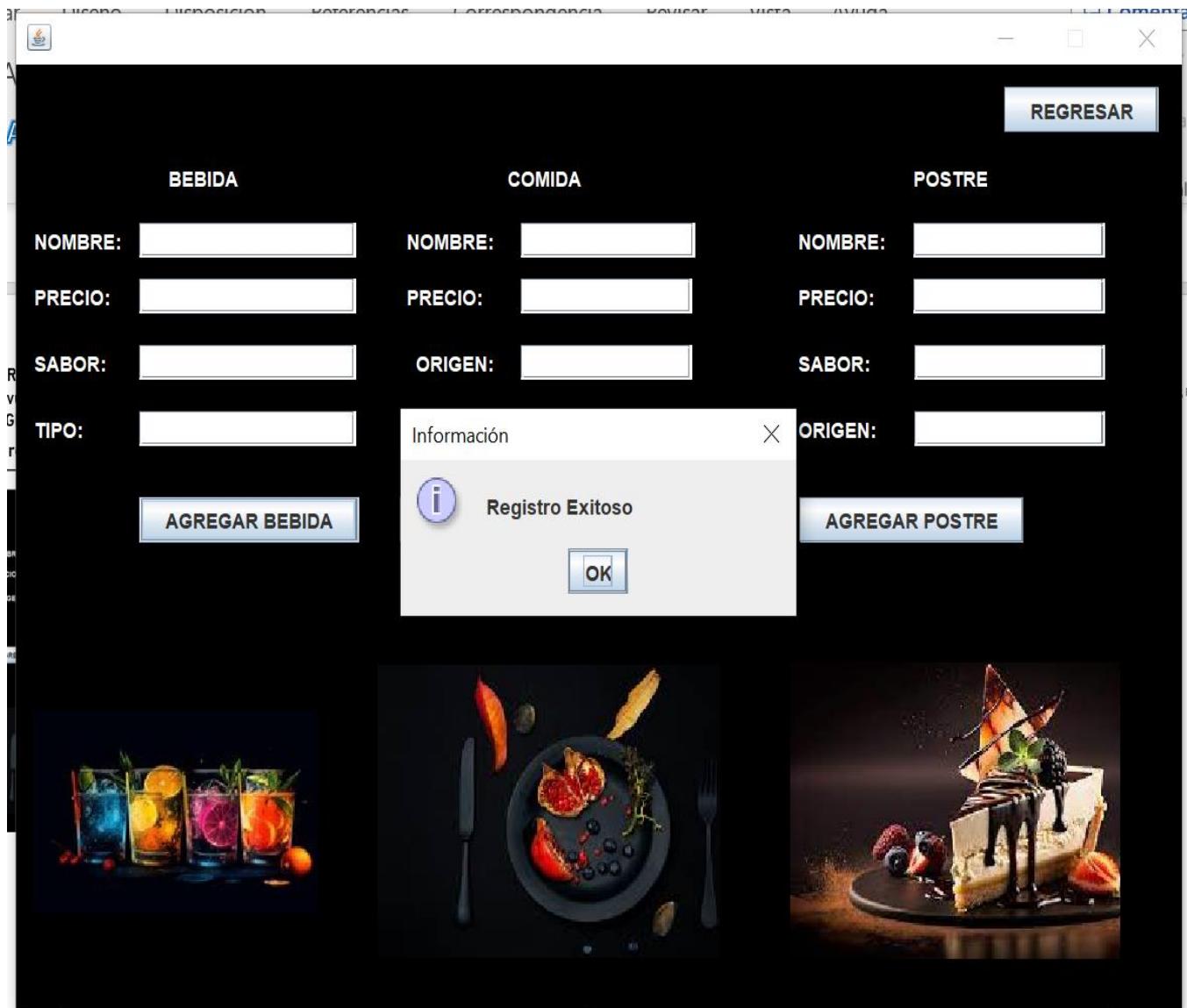
Primero registramos una bebida. Ingresamos los datos y le damos al botón de “AGREGAR BEBIDA”

The screenshot shows a Windows application window with a dark theme. At the top right are standard window controls (minimize, maximize, close). A button labeled "REGRESAR" is in the top right corner of the main area. The main area contains three horizontal rows of input fields. The first row is labeled "BEBIDA" and contains four fields: "NOMBRE" (Coca cola), "PRECIO" (18), "SABOR" (Dulce), and "TIPO" (Refresco). The second row is labeled "COMIDA" and contains two empty fields: "NOMBRE" and "PRECIO". The third row is labeled "POSTRE" and contains four empty fields: "NOMBRE", "PRECIO", "SABOR", and "ORIGEN". Below these rows are three buttons: "AGREGAR BEBIDA" (highlighted in blue), "AGREGAR COMIDA", and "AGREGAR POSTRE". At the bottom of the window are three decorative images: a row of colorful cocktails on the left, a plate of fruit in the center, and a dessert on the right.

BEBIDA		COMIDA		POSTRE	
NOMBRE:	Coca cola	NOMBRE:		NOMBRE:	
PRECIO:	18	PRECIO:		PRECIO:	
SABOR:	Dulce	ORIGEN:		SABOR:	
TIPO:	Refresco			ORIGEN:	

AGREGAR BEBIDA AGREGAR COMIDA AGREGAR POSTRE

Restaurante - Apache la Marca De

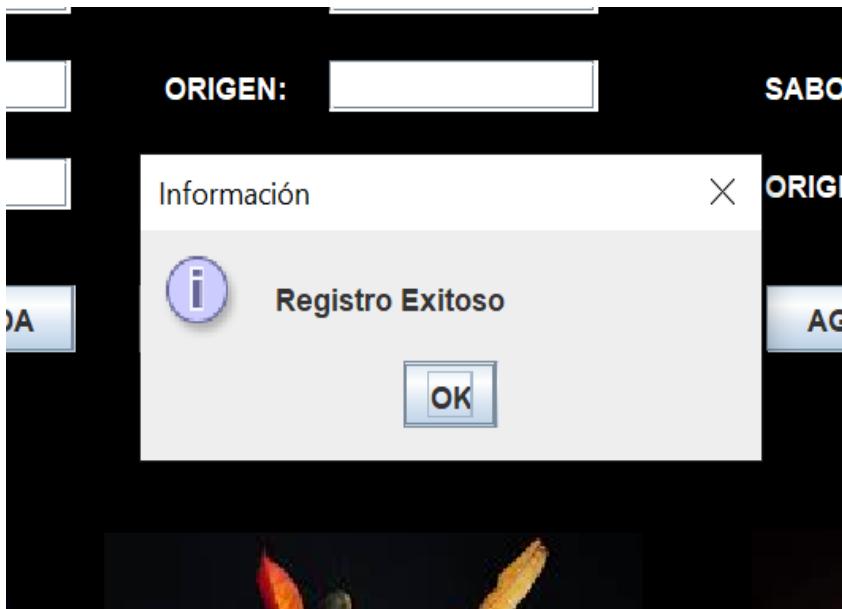


Repetimos el mismo proceso, pero esta vez para registrar una comida y finalmente un postre.

REGRESAR

BEBIDA	COMIDA	POSTRE
NOMBRE: <input type="text"/>	NOMBRE: Sopa <input type="text"/>	NOMBRE: <input type="text"/>
PRECIO: <input type="text"/>	PRECIO: 50 <input type="text"/>	PRECIO: <input type="text"/>
SABOR: <input type="text"/>	ORIGEN: Vegetal <input type="text"/>	SABOR: <input type="text"/>
TIPO: <input type="text"/>		ORIGEN: <input type="text"/>

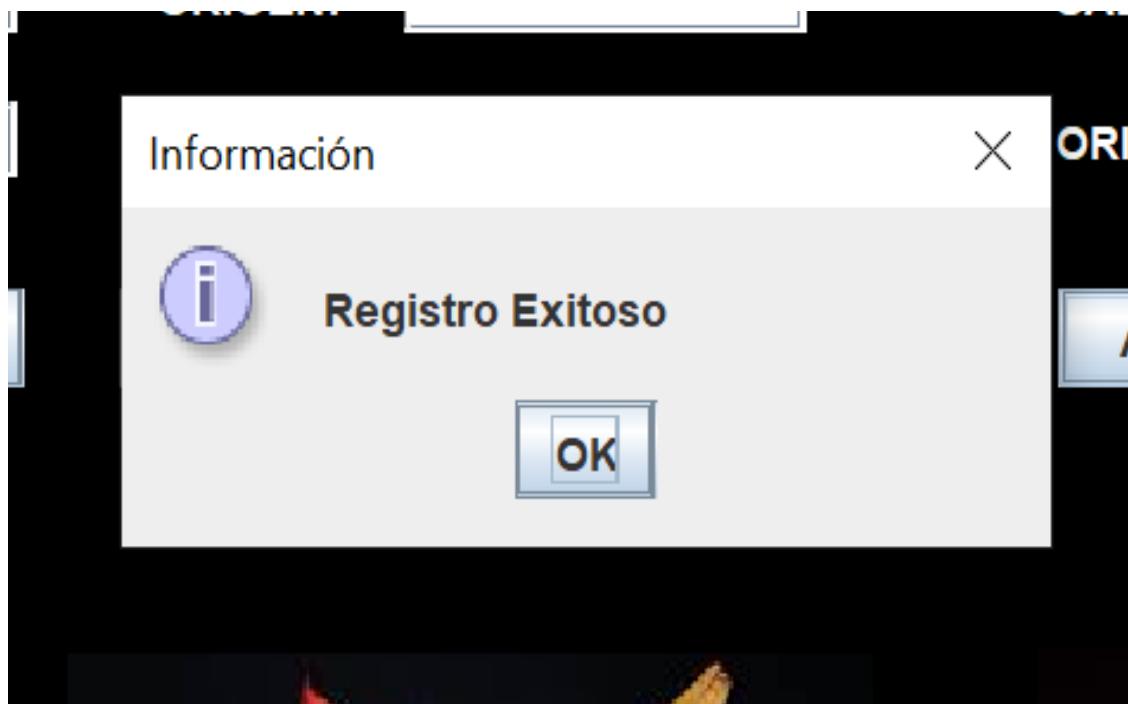
AGREGAR BEBIDA **AGREGAR COMIDA** **AGREGAR POSTRE**



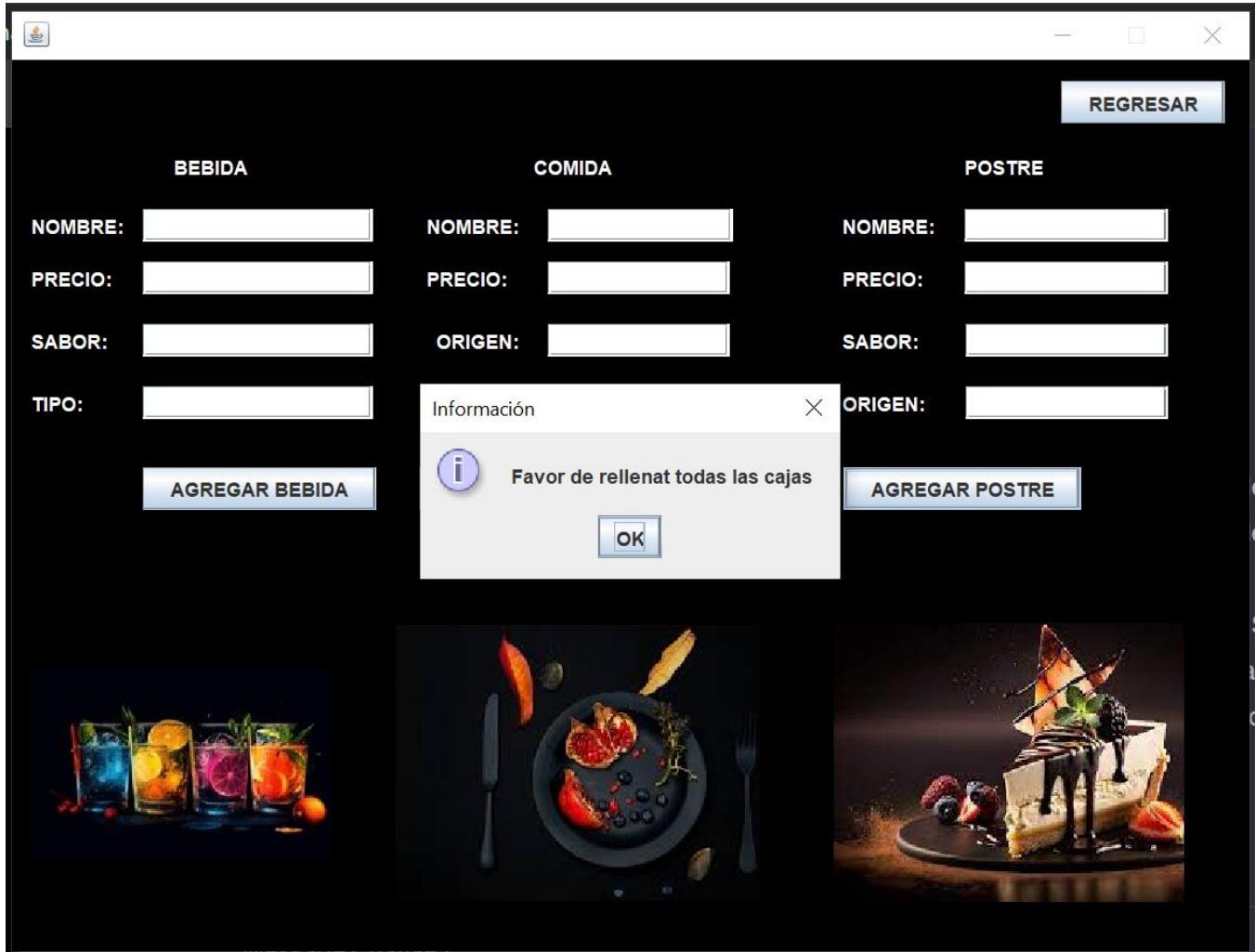
REGRESAR

BEBIDA	COMIDA	POSTRE
NOMBRE: <input type="text"/>	NOMBRE: <input type="text"/>	NOMBRE: Hotcakes <input type="text"/>
PRECIO: <input type="text"/>	PRECIO: <input type="text"/>	PRECIO: 40 <input type="text"/>
SABOR: <input type="text"/>	ORIGEN: <input type="text"/>	SABOR: Dulce <input type="text"/>
TIPO: <input type="text"/>		ORIGEN: Vegetal <input type="text"/>

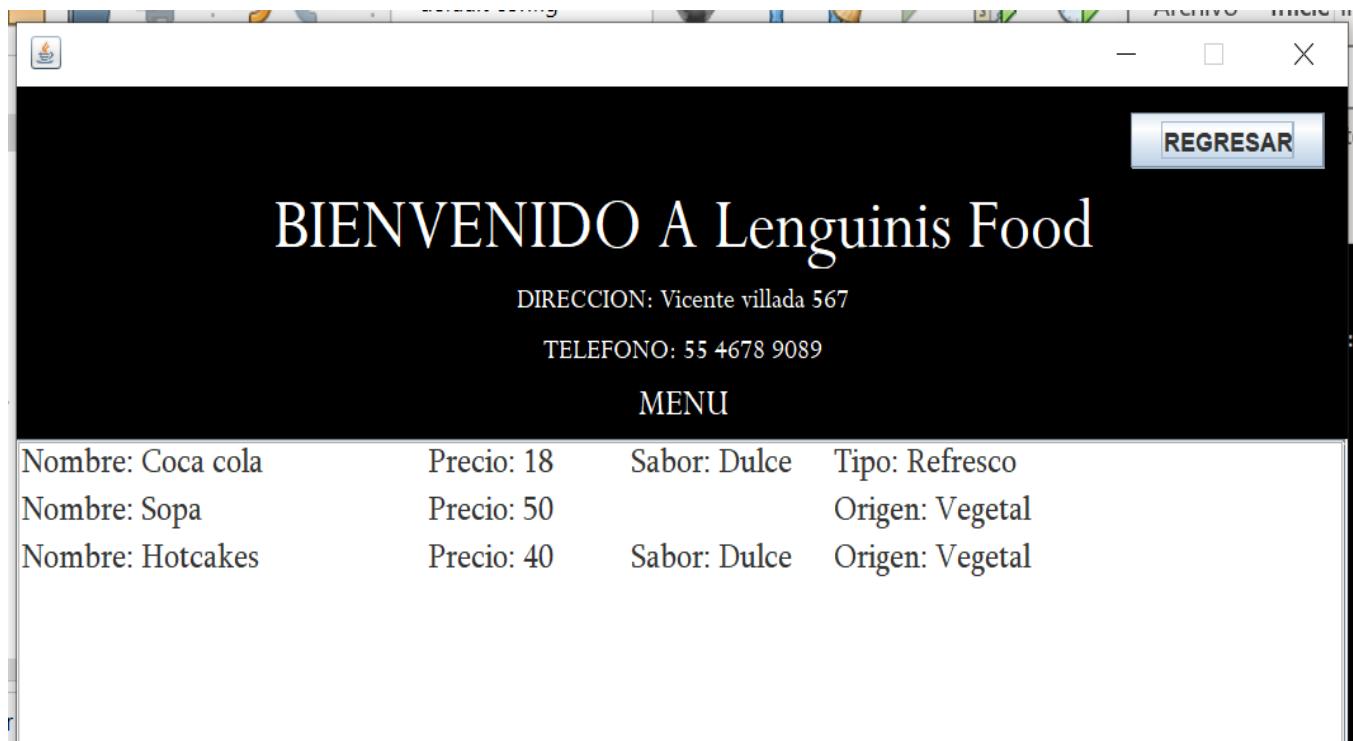
AGREGAR BEBIDA AGREGAR COMIDA AGREGAR POSTRE



En caso de querer registrar algún platillo sin llenar alguna o todas las cajas de texto se muestra:



Ya registrados los 3 platillos hacemos clic en el botón "REGRESAR" y elegimos el botón del menu principal "MOSTRAR MENU".



Después regresamos al menu principal y elegimos el botón se
“REGISTRAR REPARTIDORES”



Llenamos todos los campos para registrar a un nuevo
repartidor.

NOMBRE: Fernando
DIRECCION: Aguila Negra 675
EDAD: 19 SCORE: 9.4
MEDIO DE TRANSPORTE: MOTO
TIEMPO DE ESPERA: 10 min
AGREGAR
No hay repartidores registrados



NOMBRE: DIRECCION: EDAD: SCORE:
MEDIO DE TRANSPORTE: TIEMPO DE ESPERA:
AGREGAR
Nombre: Fernando Dirección: Aguila Negra 675 Edad: 19 Medio de transporte: MOTO Tiemp

NOMBRE: DIRECCION: EDAD: SCORE:
MEDIO DE TRANSPORTE: TIEMPO DE ESPERA:
AGREGAR
75 Edad: 19 Medio de transporte: MOTO Tiempo de espera estimado: 10 min Score: 9.4

Pasamos a registrar otros dos repartidores.

REGRESAR

NOMBRE:

DIRECCION:

EDAD: SCORE:

MEDIO DE TRANSPORTE:

TIEMPO DE ESPERA:

AGREGAR

Nombre: Fernando	Direccion: Aguilas Negras 675	Edad: 19	Medio de transporte: MOTO	Tiemp
Nombre: Cesar	Direccion: Gallo Colorado 890	Edad: 18	Medio de transporte: CARRO	Tiempo de espera estir
Nombre: Angel	Direccion: Mañanitas 456	Edad: 20	Medio de transporte: PATINETA	Tiemp

REGRESAR

NOMBRE:

DIRECCION:

EDAD: SCORE:

MEDIO DE TRANSPORTE:

TIEMPO DE ESPERA:

AGREGAR

75	Nombre: Fernando	Direccion: Aguilas Negras 675	Edad: 19	Medio de transporte: MOTO	Tiempo de espera estimado: 10 min	Score: 9.4
	Nombre: Cesar	Direccion: Gallo Colorado 890	Edad: 18	Medio de transporte: CARRO	Tiempo de espera estimado: 15 min	Score: 9.5
	Nombre: Angel	Direccion: Mañanitas 456	Edad: 20	Medio de transporte: PATINETA	Tiempo de espera estimado: 20 min	Score: 8.5

Regresamos al menu principal y seleccionamos el botón de “REALIZAR PEDIDO”.

NOMBRE: METODO DE PAGO:

DIRECCION: PLATILLOS: ELIGE LOS PLATILLOS QUE QUIERAS, PRESIONE EL BOTON DE AGREGAR ANTES DE ...
TERMINE CON SU ORDEN, CUANDO TERMINE SU ELECCION DE PLATILLOS, PRESIONE AGREGAR PÉDIDO

EDAD:

PLATILLOS:

No hay pedidos

Llenamos todos los campos siguiendo las indicaciones para agregar un platillo al pedido.

NOMBRE: Alma Alicia METODO DE PAGO: TARJETA

DIRECCION: Texcoco 230 PLATILLOS: ELIGE LOS PLATILLOS QUE QUIERAS, PRESIONE EL BOTON DE AGREGAR ANTES DE ...
TERMINE CON SU ORDEN, CUANDO TERMINE SU ELECCION DE PLATILLOS, PRESIONE AGREGAR PÉDIDO

EDAD: 30

PLATILLOS: Hotcakes

No hay pedidos

Información

El platillo o bebida agregada

Elegimos agregar “Hotcakes” al pedido por lo que ponemos el nombre del platillo y seleccionamos el botón “AGREGAR PLATILLO”

Agregamos la “Coca cola” al pedido siguiendo el mismo proceso.



En caso de ingresar un platillo que no esta registrado en el menu del restaurante se mostrara:



Ya finalizado el pedido, seleccionamos el botón “AGREGAR PEDIDO”.

NOMBRE: Alma Alicia METODO DE PAGO: TARJETA

DIRECCION: Texcoco 230 PLATILLOS: ELIGE LOS PLATILLOS QUE QUIERAS, PRESIONE EL BOTON DE AGREGAR ANTES DE ...
TERMINE CON SU ORDEN, CUANDO TERMINE SU ELECCION DE PLATILLOS, PRESIONE AGREGAR PÉDIDO

EDAD: 30

AGREGAR PEDIDO PLATILLOS: AGREGAR PLATILLO

No hay pedidos

Información

Pedido agregado

OK

NOMBRE: METODO DE PAGO: TARJETA

DIRECCION: PLATILLOS: ELIGE LOS PLATILLOS QUE QUIERAS, PRESIONE EL BOTON DE AGREGAR ANTES DE ...
TERMINE CON SU ORDEN, CUANDO TERMINE SU ELECCION DE PLATILLOS, PRESIONE AGREGAR PÉDIDO

EDAD:

AGREGAR PEDIDO PLATILLOS: AGREGAR PLATILLO

Numero de pedido: 1 Cliente: Alma Alicia Repartidor: Cesar Metodo de pago: TARJETA Platillo: Hotcakes

NOMBRE: METODO DE PAGO: TARJETA

DIRECCION: PLATILLOS: ELIGE LOS PLATILLOS QUE QUIERAS, PRESIONE EL BOTON DE AGREGAR ANTES DE ...
TERMINE CON SU ORDEN, CUANDO TERMINE SU ELECCION DE PLATILLOS, PRESIONE AGREGAR PÉDIDO

EDAD:

AGREGAR PEDIDO PLATILLOS: AGREGAR PLATILLO

Repartidor: Cesar Metodo de pago: TARJETA Platillo: Hotcakes Platillo: Coca cola Total: 58

Realizamos otro pedido para un cliente diferente.

NOMBRE: Andrea METODO DE PAGO: EFECTIVO

DIRECCION: Madrugada 550 PLATILLOS: ELIGE LOS PLATILLOS QUE QUIERAS, PRESIONE EL BOTON DE AGREGAR ANTES DE ...
TERMINE CON SU ORDEN, CUANDO TERMINE SU ELECCION DE PLATILLOS, PRESIONE AGREGAR PÉDIDO

EDAD: 22

AGREGAR PEDIDO PLATILLOS: Sopa AGREGAR PLATILLO

Repartidor: Cesar Método de pago: TARJETA Platillo: Hotcakes Platillo: Coca cola Total: 58

Información X
i El platillo o bebida agregado OK

NOMBRE: Andrea METODO DE PAGO: EFECTIVO

DIRECCION: Madrugada 550 PLATILLOS: ELIGE LOS PLATILLOS QUE QUIERAS, PRESIONE EL BOTON DE AGREGAR ANTES DE ...
TERMINE CON SU ORDEN, CUANDO TERMINE SU ELECCION DE PLATILLOS, PRESIONE AGREGAR PÉDIDO

EDAD: 22

AGREGAR PEDIDO PLATILLOS: AGREGAR PLATILLO

Repartidor: Cesar Método de pago: TARJETA Platillo: Hotcakes Platillo: Coca cola Total: 58

Información X
i Pedido agregado OK

NOMBRE: METODO DE PAGO: EFECTIVO

DIRECCION: PLATILLOS: ELIGE LOS PLATILLOS QUE QUIERAS, PRESIONE EL BOTON DE AGREGAR ANTES DE ...
TERMINE CON SU ORDEN, CUANDO TERMINE SU ELECCION DE PLATILLOS, PRESIONE AGREGAR PÉDIDO

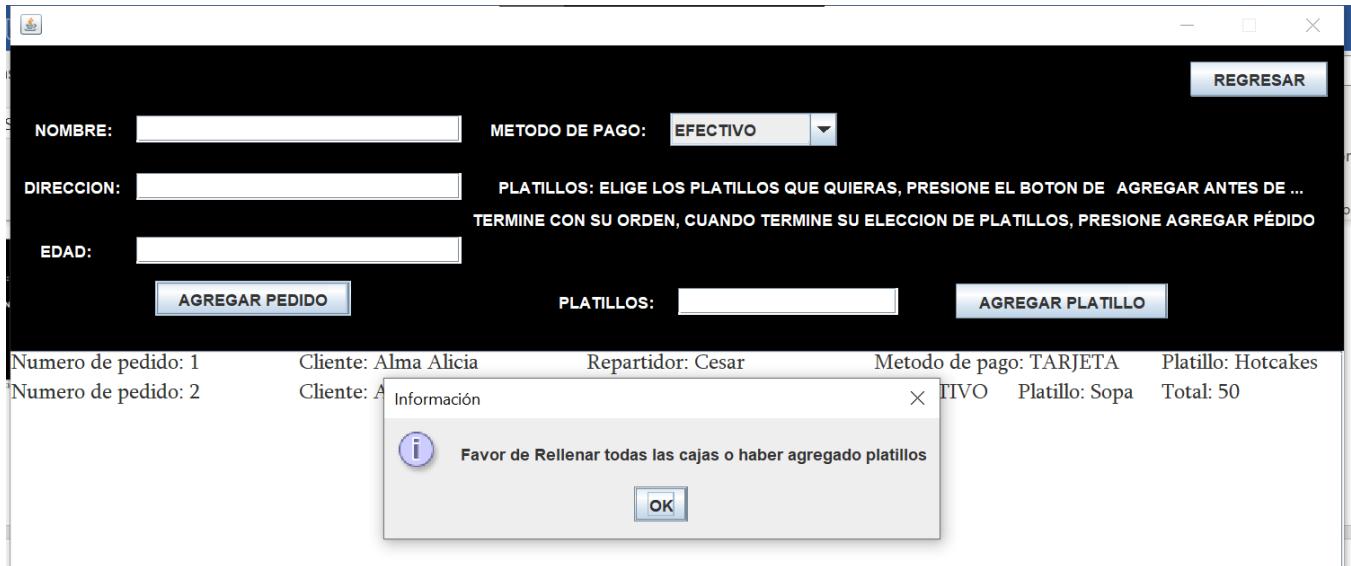
EDAD:

AGREGAR PEDIDO PLATILLOS: AGREGAR PLATILLO

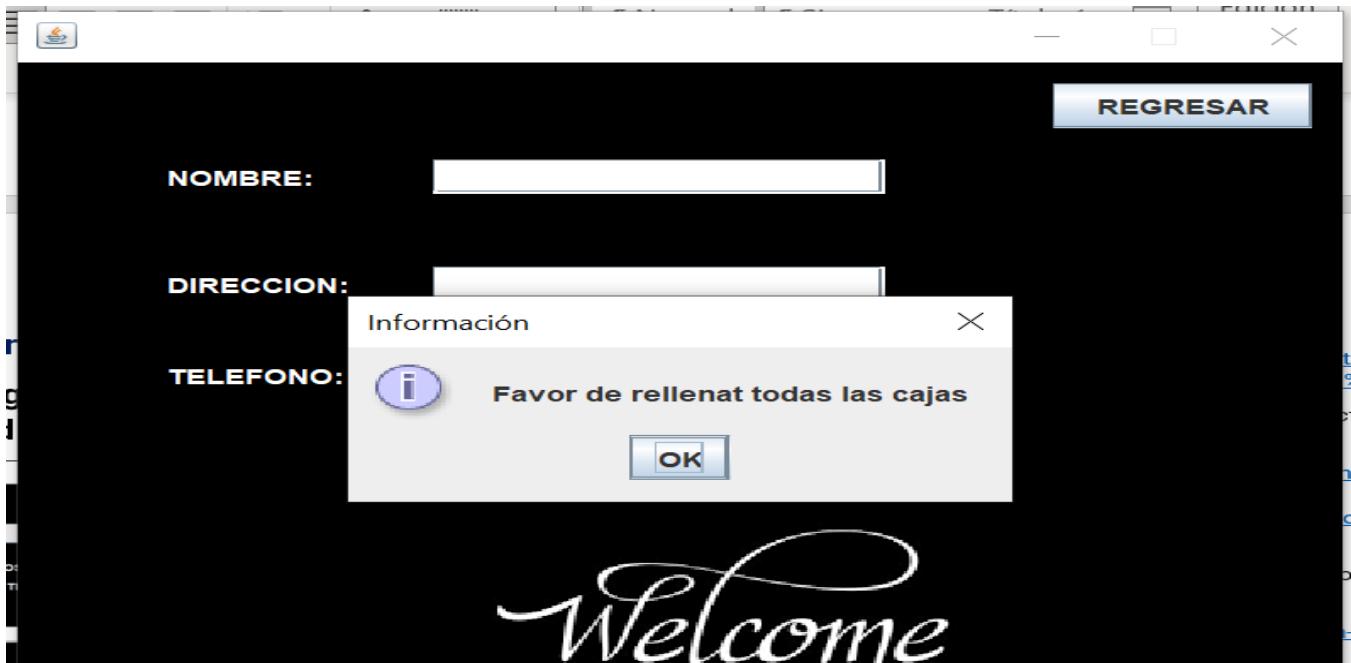
Numero de pedido: 1 Cliente: Alma Alicia Repartidor: Cesar Método de pago: TARJETA Platillo: Hotcakes
Numero de pedido: 2 Cliente: Andrea Repartidor: Fernando Método de pago: EFECTIVO Platillo: Sopa Total: 50

Mensajes alternativos.

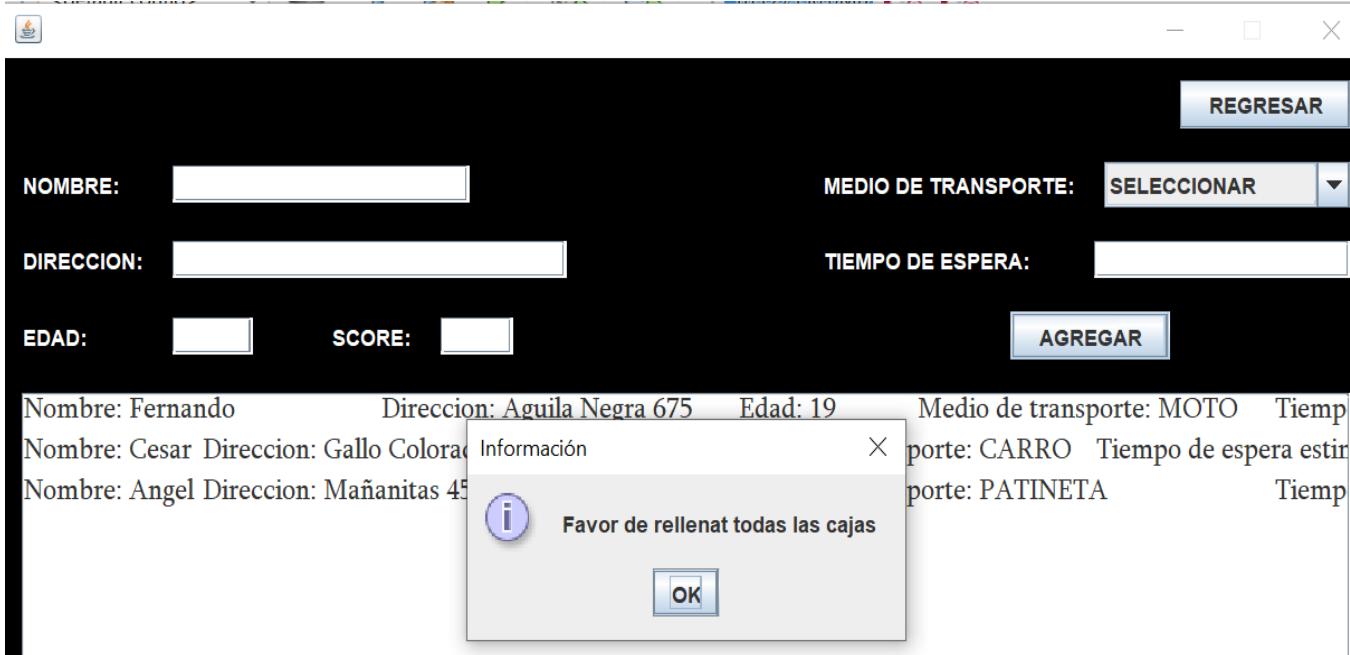
En caso de no haber rellenado alguno de las cajas de texto para agregar un pedido se muestra:



En caso de no haber rellenado alguno de las cajas de texto para registrar un restaurante se muestra:



En caso de no haber llenado alguno de las cajas de texto para registrar un repartidor se muestra:



En caso de no haber llenado la caja de texto para agregar un platillo al pedido:



Conclusiones:

Mendoza Segura Fernando

El proyecto me resultó muy entretenido ya que se pusieron en práctica todos los conocimientos vistos en la clase de paradigmas e inclusive aprendí a utilizar la interfaz gráfica de NetBeans. Es muy importante tener en claro que para poder realizar el trabajo debemos saber cómo aplicar relaciones entre clases ya que son parte esencial del proyecto pues se hace uso de la agregación, composición y herencia e igualmente se integran conceptos como el uso de interfaces y clases abstractas. Lo demás es la definición de atributos, métodos get, set, declaración de constructores y métodos alternos que realizan operaciones diferentes a los ya conocidos como por ejemplo los métodos de impresión de datos. Una vez en el aspecto gráfico tenemos que aprender a primero diseñar con elementos gráficos las ventanas a mostrar para después en el código relacionar las acciones mediante el uso de las librerías específicas y destinadas para el control y la orientación a eventos. Es muy importante siempre tener en cuenta el flujo del programa ya que así es como el main principal se irá ejecutando e irá realizando las operaciones. Fue laborioso comprender parte de la sintaxis, pero una vez comprendida se aclararon las acciones a tomar para codificar la aplicación, me gustó mucho la actividad y realmente pone en práctica todo lo visto en clase.

Cesar Said Vázquez Blancas

La conclusión de este proyecto final en Programación Orientada a Objetos (POO) en Java ha representado una inmersión completa en el diseño y desarrollo de sistemas complejos. La implementación de conceptos fundamentales como clases, herencia, composición y agregación ha permitido adquirir una comprensión más profunda de cómo estructurar y organizar el código para lograr una arquitectura robusta y modular.

La utilización de clases se reveló como una herramienta esencial para encapsular comportamientos y atributos relacionados, proporcionando una forma eficiente de modelar objetos del mundo real. La herencia, por otro lado, demostró ser valiosa al permitir la creación de jerarquías de clases que compartían características comunes, fomentando la reutilización del código y facilitando la comprensión del diseño.

La composición y la agregación añadieron otra capa de complejidad al proyecto. La capacidad de construir objetos más complejos a partir de componentes más simples mediante la composición, así como la posibilidad de relacionar objetos de manera más dinámica mediante la agregación, ampliaron las opciones de diseño y proporcionaron soluciones más flexibles.

La implementación de interfaces enriqueció significativamente la versatilidad del sistema. Al abstraer las implementaciones concretas, se logró una mayor flexibilidad y adaptabilidad del código, permitiendo una fácil extensión del sistema con nuevas funcionalidades sin afectar las partes existentes.

La incorporación de una interfaz gráfica de usuario (GUI) fue un paso crucial en la evolución del proyecto. La creación de interfaces visuales intuitivas no solo mejoró la experiencia del usuario, sino que también introdujo desafíos adicionales en términos de manejo de eventos, gestión de la interfaz y sincronización.

con la lógica del programa. Este aspecto del desarrollo subrayó la importancia de equilibrar la estética con la funcionalidad.

Las listas fueron una herramienta esencial para gestionar colecciones de datos. La selección de la estructura de lista adecuada dependió de los requisitos específicos de cada situación, destacando la importancia de elegir la herramienta adecuada para la tarea en cuestión y considerar las implicaciones de rendimiento.

A lo largo del desarrollo en NetBeans, se apreció la eficacia de un entorno integrado que facilitó la creación, depuración y prueba del código. La herramienta proporcionó una interfaz intuitiva para la gestión de proyectos y la navegación entre archivos, lo que contribuyó a una mayor productividad.

En conclusión, este proyecto ha sido una oportunidad invaluable para aplicar y consolidar los conocimientos adquiridos en el ámbito de la programación orientada a objetos. La combinación de conceptos avanzados de POO en Java, junto con la implementación práctica en un entorno de desarrollo profesional, ha proporcionado una experiencia integral que ha fortalecido la comprensión de la programación orientada a objetos.

Bibliografía:

IV – Sobrecarga de métodos y constructores, Ing. Carlos Alberto Román Zamitz, Facultad de Ingeniería, UNAM, Recuperado el 14 de noviembre del 2023, de

http://profesores.fi-b.unam.mx/carlos/java/java_basico4_6.html#:~:text=La%20sobrecarga%20de%20m%C3%A9todos%20es,cu%C3%A1l%20definici%C3%B3n%20de%20m%C3%A9todo%20ejecutar.

Videotutorial Sobrecarga del constructor: destructor. - C# - LinkedIn, linkedin.com, Recuperado el 14 de noviembre del 2023, de

<https://es.linkedin.com/learning/c-sharp-programacion-orientada-a-objetos/sobrecarga-del-constructor-destructor#:~:text=La%20sobrecarga%20en%20los%20constructores,%22Estudiante%22%20que%20reclama%20par%C3%A1metros>

Herencia en Java: definición y ejemplos, María Coppola, HubSpot, Recuperado el 1 de Diciembre del 2023 de,

<https://blog.hubspot.es/website/que-es-herencia-java>

¿Para que sirve la línea @Override en java?, E. Betanzos, stackoverflow, Recuperado el 1 de Diciembre del 2023, de

<https://es.stackoverflow.com/>

Relaciones entre clases, Microsoft, Recuperado el 2 de Diciembre de 2023, de

https://www.unirioja.es/cu/jearansa/0910/archivos/EIPR_Tema02.pdf

Herencia y polimorfismo, Michael González Harbour, Mario Aldea Rivas, Universidad de Cantabria, Recuperado el 2 de diciembre de 2023, de

<https://ocw.unican.es/pluginfile.php/2330/course/section/2281/cap8-herencia.pdf>

SOBREESCRIBIR MÉTODOS EN JAVA. TIPO ESTÁTICO Y DINÁMICO. LIGADURA. MÉTODOS POLIMÓRFICOS; Alex Rodríguez, Curso “Aprender programación Java desde cero”, Recuperado el 2 de Diciembre de 2023, de

<https://www.aprenderaprogramar.com/attachments/article/660/CU00690B%20sobreescritura%20metodos%20java%20estatico%20dinamico%20ligadura%20compilador.pdf>

Clases abstractas e interfaces en java, Universidad de la Rioja, Recuperado el 2 de Diciembre de 2023, de

https://www.unirioja.es/cu/jearansa/0910/archivos/EIPR_Tema04.pdf

Estructura de Datos Colecciones, Instituto tecnológico de Zacatecas, Recuperado el 2 de Diciembre de 2023, de

<http://mapaches.itz.edu.mx/~mbarajas/edinf/Colecciones.pdf>

Programación Orientada a Objetos con Java, Universidad Politécnica de Madrid, Recuperado el 2 de Diciembre de 2023, de

https://www.etsisi.upm.es/sites/default/files/curso_2013_14/MASTER/MIW.JEE.PO0J.pdf

Creación de interfaces gráficas, Facultad: Ingeniería Escuela: Computación Asignatura: Java Avanzado, Universidad Don Bosco, Recuperado el 2 de Diciembre de 2023, de

https://www.udb.edu.sv/udb_files/recursos_guias/informatica-ingeneria/java-avanzado/2019/i/guia-5.pdf

Interfaces de usuario con Netbeans, Pedro Corcuer Dpto. Matemática Aplicada y Ciencias de la Computación Universidad de Cantabria, Recuperado el 2 de Diciembre de 2023, de

https://personales.unican.es/corcuerp/java/slides/guis_netbeans.pdf