Manual Técnico

Gestión de Delivery

Fecha: 06 de octubre de 2023

Contenido

1. OBJETIVO

El objetivo de este manual técnico es proporcionar a cualquier persona interesada en el sistema del restaurante de comida rápida una guía completa sobre la estructura, funcionamiento y desarrollo de la aplicación de gestión de delivery. Este manual tiene como propósito facilitar la comprensión de cómo se ha implementado la aplicación, permitiendo su mantenimiento, extensión y personalización de acuerdo con las necesidades cambiantes del usuario.

2. ALCANCE

Gestión de Delivery – Es una aplicación que facilita la administración, registro, control y manejo de productos, pedidos, personal repartidor y respaldos de los pedidos enviados, dado que posee una interfaz gráfica, se describirán sus funciones y demás detalles relacionados al código, los recursos y paquetes utilizados.

3. DESARROLLO DEL MANUAL TÉCNICO

3.1 Arquitectura de la Aplicación

3.1.1 Descripción General

La aplicación Gestión de Delivery del Restaurante de Comida Rápida fue desarrollada con el fin de gestionar la creación y envío de pedidos a los del restaurante. La aplicación fue desarrollada en Java aplicando un lenguaje de Programación Orientada a Objetos (POO) e hilos para visualizar la trayectoria de los repartidores, lo cual facilita tener un mayor control sobre los mismos; contando con clases y clases de interfaz gráfica con las bibliotecas “GUI” de Java como Swing.

3.1.2 Componentes Principales

Se utilizaron clases como Producto, Motociclista, Orden, AppState. Mientras que para la interfaz gráfica se utilizaron JFrame Form´s siendo estas: RestauranteMainFrame y NuevoProductoJFrame; JPanel´s como FastFoodPanel, PapasPanel, PizzaPanel, PolloPanel y GaseosaPanel; y esencialmente para ejecutar los hilos se creó la clase MovimientoImagen la cual posee dos imágenes, una como fondo y otra como simulador del personal de delivery y un panel que las contiene.

3.1.3 Diagrama de Clases

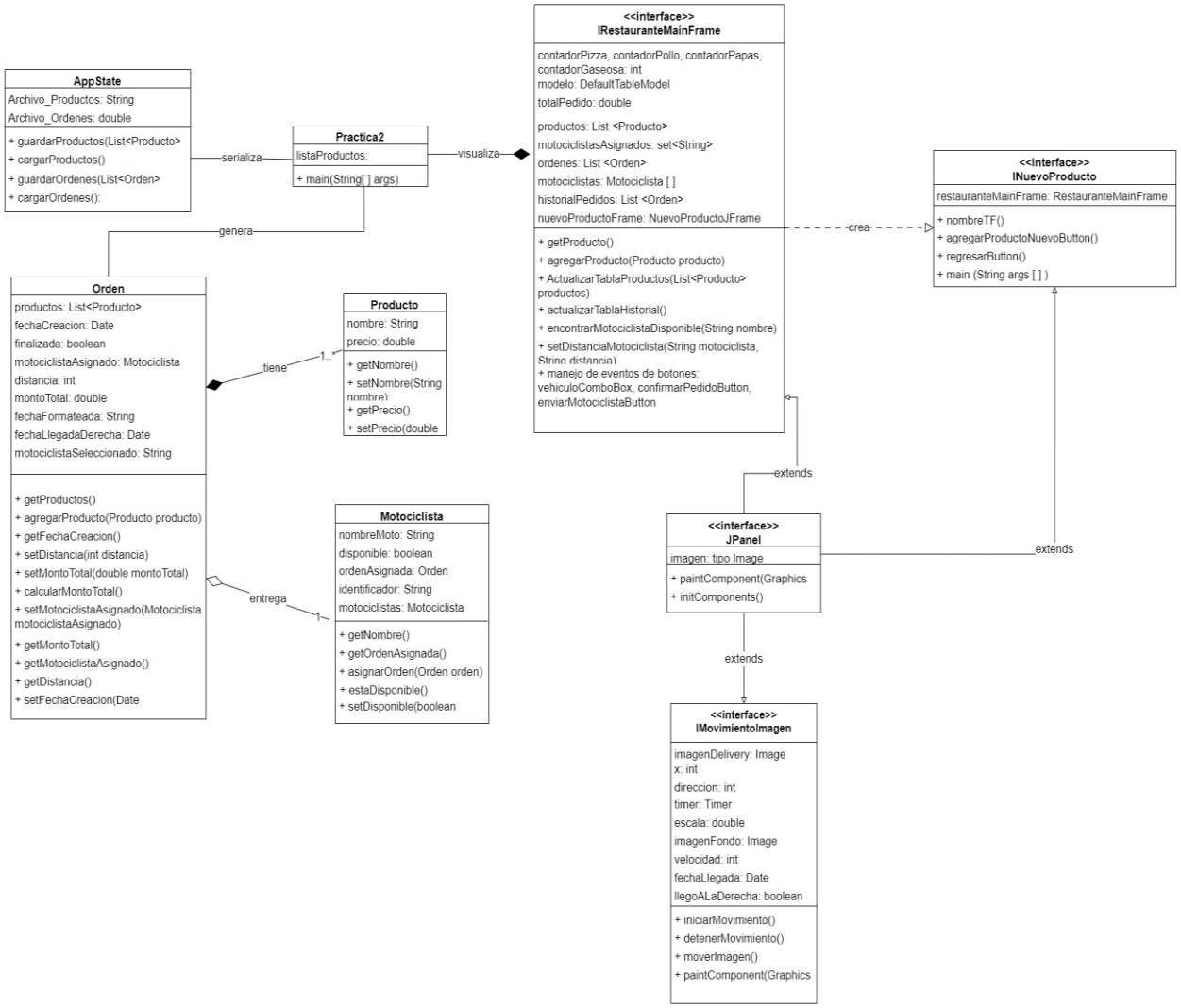

Diagrama 1: Elaboración propia, 2023

3.2 Estructura del Proyecto

3.2.1 Paquetes Principales

src/com/mycompany/practica2: Paquete principal que contiene clases relacionadas con la aplicación.

src/com/mycompany/practica2.gui: Paquete principal que contiene las clases relacionadas a la interfaz gráfica.

3.2.2 Diccionario de Clases

3.2.2.1 Practica2

Descripción:

La clase Practica2 es la clase principal de la aplicación que gestiona un restaurante. Contiene una lista de productos disponibles en el restaurante y crea la interfaz de usuario principal para interactuar con el restaurante.

Atributos:

- listaProductos (Tipo: List<Producto>): Una lista que almacena objetos de la clase Producto. Contiene los productos disponibles en el restaurante.

Métodos:

- main(String[] args): El método principal de la aplicación. Se ejecuta al iniciar la aplicación y realiza las siguientes acciones:
  - Crea una lista de productos y la inicializa con algunos productos de ejemplo.
  - Crea una instancia de RestauranteMainFrame, que es la ventana principal de la interfaz de usuario del restaurante.

- Hace visible la ventana principal.

Relaciones:

La clase Practica2 utiliza la clase Producto para representar los productos disponibles en el restaurante.

Ejemplos de Uso:

```
public class Practica2 {
    public static List<Producto> listaProductos = new ArrayList<>();
    public static void main(String[] args) {
        listaProductos.add(new Producto("Pizza", 55.00));
        listaProductos.add(new Producto("Pollo Frito", 130.00));
        listaProductos.add(new Producto("Papas Fritas", 15.00));
        listaProductos.add(new Producto("Gaseosa", 12.00));
        RestauranteMainFrame mainFrame = new RestauranteMainFrame();
        mainFrame.setVisible(true);
    }
}
```

3.2.2.2 Producto

Descripción:

La clase Producto representa un elemento dentro de un sistema, que generalmente se utiliza para describir un artículo en un contexto de tienda o restaurante. Cada producto tiene un nombre y un precio asociado.

Atributos:

- nombre (Tipo: String): El nombre del producto.
- precio (Tipo: double): El precio del producto.

Constructores:

- Producto(String nombre, double precio): Constructor que crea una instancia de la clase Producto con un nombre y un precio específicos. Se utiliza para inicializar un producto con valores iniciales.

Métodos:

- getNombre(): Método que devuelve el nombre del producto.
- setNombre(String nombre): Método que permite establecer el nombre del producto.
- getPrecio(): Método que devuelve el precio del producto.
- setPrecio(double precio): Método que permite establecer el precio del producto.

Interfaz:

- ProductoAgregadoListener: Una interfaz que define un método productoAgregado(Producto producto) que puede ser implementado por clases que desean ser notificadas cuando se agrega un producto. Esta interfaz se utiliza para permitir la observación de eventos relacionados con productos.

Ejemplos de Uso:

```
public class Producto implements Serializable {

    public String nombre;
    public double precio;
```

```
    public Producto(String nombre, double precio) {
        this.nombre = nombre;
        this.precio = precio;
    }
    public String getNombre() {
        return nombre;
    }

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

    public double getPrecio() {
        return precio;
    }

    public void setPrecio(double precio) {
        this.precio = precio;
    }
    public interface ProductoAgregadoListener {
        void productoAgregado(Producto producto);
    }
}
```

3.2.2.3 Orden

Descripción:

La clase Orden representa una orden o solicitud en un sistema de delivery del restaurante. Contiene información sobre los productos pedidos, la fecha de creación

de la orden, el estado de finalización, el motociclista asignado, la distancia de entrega, el monto total, y otros detalles relacionados con la orden.

Atributos:

- productos (Tipo: List<Producto>): Una lista que almacena objetos de la clase Producto. Representa los productos incluidos en la orden.
- fechaCreacion (Tipo: Date): La fecha y hora en que se creó la orden.
- finalizada (Tipo: boolean): Un indicador que muestra si la orden ha sido finalizada o no.
- motociclistaAsignado (Tipo: Motociclista): Una referencia al motociclista asignado a la entrega de la orden.
- distancia (Tipo: int): La distancia de entrega de la orden.
- montoTotal (Tipo: double): El monto total de la orden.
- fechaFormateada (Tipo: String): Una cadena formateada de fecha (puede ser una representación legible para humanos de la fecha).
- fechaLlegadaDerecha (Tipo: Date): La fecha de llegada prevista de la orden.
- motociclistaSeleccionado (Tipo: String): El nombre del motociclista seleccionado para la entrega.

Constructores:

- Orden (): Constructor que inicializa una instancia de la clase Orden con valores predeterminados para algunos de sus atributos.

Métodos:

- getProductos(): Método que devuelve la lista de productos en la orden.

- agregarProducto(Producto producto): Método que agrega un producto a la lista de productos de la orden.
- getFechaCreacion(): Método que devuelve la fecha de creación de la orden.
- setFechaCreacion(Date fechaCreacion): Método que permite establecer la fecha de creación de la orden.
- getMotociclistaAsignado(): Método que devuelve el motociclista asignado a la orden.
- setMotociclistaAsignado(Motociclista motociclistaAsignado): Método que permite establecer el motociclista asignado a la orden.
- getDistancia(): Método que devuelve la distancia de entrega de la orden.
- setDistancia(int distancia): Método que permite establecer la distancia de entrega de la orden.
- getMontoTotal(): Método que devuelve el monto total de la orden.
- calcularMontoTotal(): Método que calcula el monto total de la orden en función de los productos incluidos.
- setFechaFormateada(String fechaFormateada): Método que permite establecer una representación formateada de la fecha.
- getFechaLlegadaDerecha(): Método que devuelve la fecha de llegada prevista de la orden.
- setFechaLlegadaDerecha(Date fechaLlegadaDerecha): Método que permite establecer la fecha de llegada prevista de la orden.
- setFechaEntrega(Date fechaEntrega): Método que permite establecer la fecha de entrega de la orden.

- setMontoTotal(double montoTotal): Método que permite establecer el monto total de la orden.
- getMotociclistaSeleccionado(): Método que devuelve el nombre del motociclista seleccionado.
- setMotociclistaSeleccionado(String motociclistaSeleccionado): Método que permite establecer el nombre del motociclista seleccionado.
- toString(): Método que devuelve una representación en cadena de la orden, incluyendo el motociclista asignado, la distancia, los productos, la fecha de creación y la fecha de llegada prevista.

Ejemplos de Uso:

```
public class Orden implements Serializable {
    public List<Producto> productos;
    public Date fechaCreacion;
    private boolean finalizada;
    private Motociclista motociclistaAsignado;
    private int distancia;
    private double montoTotal;
    public String fechaFormateada;
    private Date fechaLlegadaDerecha;
    private String motociclistaSeleccionado;

    public Orden() {
        productos = new ArrayList<>();
```

```
        fechaCreacion = new Date();
        finalizada = false;
        motociclistaAsignado = null;
        distancia = 0;
        montoTotal = 0.0;
    }
    public List<Producto> getProductos() {
        return productos;
    }
    public void agregarProducto(Producto producto) {
        productos.add(producto);
    }

    public Date getFechaCreacion() {
        return fechaCreacion;
    }
    public void setFechaCreacion(Date fechaCreacion) {
        this.fechaCreacion = fechaCreacion;
    }

    public Motociclista getMotociclistaAsignado() {
        return motociclistaAsignado;
    }
```

```
    public        void        setMotociclistaAsignado(Motociclista
motociclistaAsignado) {
        this.motociclistaAsignado = motociclistaAsignado;
    }
    public int getDistancia() {
        return distancia;
    }

    public void setDistancia(int distancia) {
        this.distancia = distancia;
    }
    public double getMontoTotal() {
        return montoTotal;
    }
    public void calcularMontoTotal() {
        montoTotal = 0.0;
        for (Producto producto : productos) {
            montoTotal += producto.getPrecio();
        }
        System.out.println("Se calcula el monto total");
    }
    public void setFechaFormateada(String fechaFormateada) {
        this.fechaFormateada = fechaFormateada;
    }
```

```
    public Date getFechaLlegadaDerecha() {
        return fechaLlegadaDerecha;
    }
    public void setFechaLlegadaDerecha(Date fechaLlegadaDerecha) {
        this.fechaLlegadaDerecha = fechaLlegadaDerecha;
    }
    public void setFechaEntrega(Date fechaEntrega) {
        this.fechaLlegadaDerecha = fechaEntrega;
    }
    public void setMontoTotal(double montoTotal) {
        this.montoTotal = montoTotal;
    }
    public String getMotociclistaSeleccionado() {
        return motociclistaSeleccionado;
    }

    public         void         setMotociclistaSeleccionado(String
motociclistaSeleccionado) {
        this.motociclistaSeleccionado = motociclistaSeleccionado;
    }
    public String toString() {
        return "Moticiclista " + motociclistaAsignado + ", Distancia
" + distancia + ", productos" + productos + "fecha de creacion" +
fechaCreacion + "fecha de entrega " + fechaLlegadaDerecha;
```

```
    }
}
```

3.2.2.4 Motociclista

Descripción:

La clase Motociclista representa a un repartidor de motocicletas que se utiliza en un sistema de entregas. Contiene información sobre el nombre del motociclista, su disponibilidad, la orden que se le ha asignado y un identificador único.

Atributos:

- nombreMoto (Tipo: String): El nombre del motociclista.
- disponible (Tipo: boolean): Un indicador que muestra si el motociclista está disponible para realizar entregas o no.
- ordenAsignada (Tipo: Orden): Una referencia a la orden que se le ha asignado al motociclista.
- identificador (Tipo: String): Un identificador único para el motociclista.
- motociclistas (Tipo: Motociclista[]): Un arreglo que almacena objetos de la clase Motociclista. Puede utilizarse para gestionar múltiples motociclistas en el sistema.

Constructores:

- Motociclista(String nombre, String identificador): Constructor que crea una instancia de la clase Motociclista con un nombre y un identificador específicos, y establece su disponibilidad inicial como verdadera.

Métodos:

- getNombre(): Método que devuelve el nombre del motociclista.

- getIdentificador(): Método que devuelve el identificador único del motociclista.
- getOrdenAsignada(): Método que devuelve la orden asignada al motociclista (puede ser null si no tiene ninguna orden asignada).
- asignarOrden(Orden orden): Método que permite asignar una orden al motociclista.
- desasignarOrden(): Método que permite desasignar la orden del motociclista.
- estaDisponible(): Método que devuelve si el motociclista está disponible para realizar entregas (verdadero) o no (falso).
- setDisponible(boolean disponible): Método que permite cambiar el estado de disponibilidad del motociclista.

Ejemplos de Uso:

```
public class Motociclista implements Serializable {
    private String nombreMoto;
    private boolean disponible;
    private Orden ordenAsignada;
    private String identificador;
    public Motociclista[] motociclistas = new Motociclista[3];

    public Motociclista(String nombre, String identificador) {
        this.nombreMoto = nombre;
        this.identificador = identificador;
        this.disponible = true;
```

```
}
public String getNombre() {
    return nombreMoto;
}
public String getIdentificador() {
    return identificador;
}
public Orden getOrdenAsignada() {
    return ordenAsignada;
}
public void asignarOrden(Orden orden) {
    this.ordenAsignada = orden;
}
public void desasignarOrden() {
    this.ordenAsignada = null;
}
public boolean estaDisponible() {
    return disponible;
}
public void setDisponible(boolean disponible) {
    this.disponible = disponible;
}
```

3.2.2.5 AppState

Descripción:

La clase AppState gestiona el estado de la aplicación y proporciona métodos para cargar y guardar datos relacionados con productos y órdenes en archivos binarios.

Atributos:

- ARCHIVO_PRODUCTOS (Tipo: String): Una cadena que representa la ruta al archivo binario donde se guardarán los datos de productos.
- ARCHIVO_ORDENES (Tipo: String): Una cadena que representa la ruta al archivo binario donde se guardarán los datos de órdenes.

Métodos:

- guardarProductos(List<Producto> productos): Este método recibe una lista de objetos Producto y los guarda en un archivo binario en la ubicación especificada por ARCHIVO_PRODUCTOS. Primero, verifica si el archivo existe y, de no ser así, crea el directorio correspondiente. Luego, utiliza un ObjectOutputStream para escribir la lista de productos en el archivo binario.
- cargarProductos(): Este método carga la lista de productos desde el archivo binario especificado por ARCHIVO_PRODUCTOS y la devuelve como una lista de objetos Producto. Utiliza un ObjectInputStream para leer el archivo binario y convertirlo en una lista de productos.
- guardarOrdenes(List<Orden> ordenes): Similar al método guardarProductos, este método recibe una lista de objetos Orden y los guarda en un archivo binario en la ubicación especificada por ARCHIVO_ORDENES. También verifica si el archivo existe y, de no ser así, crea el directorio correspondiente. Utiliza un ObjectOutputStream para escribir la lista de órdenes en el archivo binario.
- cargarOrdenes(): Similar al método cargarProductos, este método carga la lista de órdenes desde el archivo binario especificado por ARCHIVO_ORDENES y la devuelve como una lista de objetos Orden. Utiliza

un ObjectInputStream para leer el archivo binario y convertirlo en una lista de órdenes.

3.2.2.6 MovimientoImagen

Descripción:

La clase MovimientoImagen es una clase de Java que extiende JPanel y se utiliza para crear una animación de movimiento de una imagen en un panel. La animación simula el movimiento de una imagen de entrega a lo largo de un camino.

Atributos:

- imagenDelivery (Tipo: Image): Representa la imagen del repartidor.
- x (Tipo: int): La posición horizontal actual de la imagen en el panel.
- direccion (Tipo: int): La dirección de movimiento de la imagen (-1 para izquierda, 1 para derecha).
- timer (Tipo: Timer): Un temporizador que controla la animación del movimiento.
- escala (Tipo: double): La escala de la imagen (tamaño relativo).
- imagenFondo (Tipo: Image): Representa la imagen de fondo del camino.
- velocidad (Tipo: int): La velocidad de movimiento de la imagen.
- fechaLlegadaDerecha (Tipo: Date): La fecha y hora en que la imagen llega al lado derecho del panel.
- llegoALaDerecha (Tipo: boolean): Un indicador que registra si la imagen ha llegado al lado derecho del panel.

Constructores:

- MovimientoImagen(): Constructor que inicializa la clase. Carga las imágenes, establece el tamaño del panel y configura el temporizador para el movimiento.

Métodos:

- setVelocidad(int velocidad): Permite establecer la velocidad de movimiento de la imagen.
- iniciarMovimiento(): Inicia la animación de movimiento si no está en marcha.
- detenerMovimiento(): Detiene la animación de movimiento si está en marcha.
- moverImagen(): Método privado que actualiza la posición de la imagen en función de la dirección y velocidad de movimiento.
- paintComponent(Graphics g): Método para dibujar la imagen y el fondo en el panel.

Ejemplos de Uso:

Esta es solamente parte del código de esta clase:

```
public MovimientoImagen() {
        try {
            imagenDelivery = ImageIO.read(new
File("C:/Users/Usuario/Downloads/delivery-man.png"));
            imagenFondo = ImageIO.read(new
File("C:/Users/Usuario/Downloads/road (1).png"));
        } catch (IOException e) {
            e.printStackTrace();
        }
```

```
        setPreferredSize(new
Dimension((int)(imagenDelivery.getWidth(this)      *      escala),
(int)(imagenDelivery.getHeight(this) * escala)));
        x = 0;
        direccion = 1;
        timer = new Timer(10, new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                moverImagen();
            }
        });
    }
public void setVelocidad(int velocidad) {
    this.velocidad = velocidad;
}
    public void iniciarMovimiento() {
        if (!timer.isRunning()) {
            x = 0; // Restablecer la posición inicial
            direccion = 1; // Restablecer la dirección
            timer.start();
        }
    }
    public void detenerMovimiento() {
        if (timer.isRunning()) {
```

```
            timer.stop();
            x = 0; // Restablece la posición inicial
            direccion = 1; // Restablece la dirección
            repaint();
        }
    }
    private void moverImagen() {
        x += velocidad * direccion;

        if (x <= 0 || x >= getWidth() - imagenDelivery.getWidth(this)
* escala) {
            direccion *= -1; // Cambiar de dirección al llegar al
borde
            // Verificar si ha completado una ida y vuelta
            if (x <= 0) {
                timer.stop(); // Detener el temporizador después de
una ida y vuelta
            }     else     if     (x     >=     getWidth()     -
imagenDelivery.getWidth(this) * escala && !llegoALaDerecha) {
                // Cuando llega al lado derecho antes de girar y no
ha llegado antes, registra la fecha y hora
                fechaLlegadaDerecha = new Date();
                llegoALaDerecha = true;
            }
```

```
        }
        repaint();
    }
}
```

3.2.2.7 RestauranteMainFrame

Descripción:

La clase RestauranteMainFrame es una clase de Java que extiende javax.swing.JFrame y se utiliza para crear la interfaz gráfica principal de una aplicación de gestión de pedidos en un restaurante.

Atributos:

- contadorPizza, contadorPollo, contadorPapas, contadorGaseosa (Tipo: int): Contadores para realizar un seguimiento de la cantidad de productos específicos agregados.
- totalPedido (Tipo: double): El monto total del pedido.
- modelo (Tipo: DefaultTableModel): El modelo de tabla para mostrar productos agregados.
- productos (Tipo: List<Producto>): Lista de productos disponibles en el restaurante.
- nuevoProductoFrame (Tipo: NuevoProductoJFrame): Una ventana para agregar nuevos productos.
- motociclistasAsignados (Tipo: Set<String>): Conjunto de nombres de motociclistas asignados a las órdenes.
- ordenes (Tipo: List<Orden>): Lista de órdenes activas.

- motociclistas (Tipo: Motociclista[]): Un arreglo de objetos Motociclista.
- historialPedidos (Tipo: List<Orden>): Lista de órdenes históricas.
- ordenActual (Tipo: Orden): La orden actual que se está gestionando.
- enviarMotociclista1Habilitado, enviarMotociclista2Habilitado, enviarMotociclista3Habilitado (Tipo: boolean): Variables que indican si se puede enviar un motociclista específico.

Constructores:

- RestauranteMainFrame(): Constructor que inicializa la interfaz gráfica del restaurante y configura los componentes visuales.

Métodos:

- getProducto(): Devuelve la lista de productos disponibles en el restaurante.
- agregarProducto(Producto producto): Agrega un producto a la lista de productos disponibles.
- getOrdenes(): Devuelve la lista de órdenes activas.
- ActualizarTablaProductos(List<Producto> productos): Actualiza la tabla de productos disponibles en la interfaz gráfica.
- abrirNuevoProductoFrame(): Abre la ventana para agregar nuevos productos.
- reiniciarContadores(): Reinicia los contadores de productos.
- encontrarMotociclistaDisponible(String nombre): Busca un motociclista disponible por nombre.
- actualizarTablaHistorial(): Actualiza la tabla de historial de pedidos.

- setDistanciaMotociclista(String motociclista, String distancia): Actualiza la distancia en la interfaz para un motociclista específico.
- Métodos de manejo de eventos de botones y componentes de la interfaz gráfica, como agregarPizzaButtonActionPerformed, agregarPolloButtonActionPerformed, agregarPapasButtonActionPerformed, agregarGaseosaButtonActionPerformed, confirmarPedidoButtonActionPerformed, agregarProductoButtonActionPerformed, vehiculoComboBoxActionPerformed, nuevoProductoButtonActionPerformed, enviarMotociclista1ButtonActionPerformed, enviarTodosButtonActionPerformed, enviarMotociclista2ButtonActionPerformed, enviarMotociclista3ButtonActionPerformed.
- Método main(String args[]): El punto de entrada principal de la aplicación, donde se crea una instancia de la clase RestauranteMainFrame y se muestra la ventana principal.

Ejemplos de Uso:

Estos son solo algunos de los eventos más importantes de esta clase, dado que el código es muy extenso:

```
private void actualizarTablaHistorial() {
    DefaultTableModel modeloHistorial = (DefaultTableModel) historialJTable.getModel();
```

```
    modeloHistorial.setRowCount(0);  //  Borra  todas  las  filas
existentes
    int columnaFecha = 4;
    SimpleDateFormat          formatoFecha          =          new
SimpleDateFormat("dd/MM/yyyy HH:mm:ss", new Locale("es", "ES"));
    for (Orden orden : historialPedidos) {
        String                   fechaFormateada                  =
formatoFecha.format(orden.getFechaCreacion());
        String              motociclistaSeleccionado              =
orden.getMotociclistaSeleccionado();  //  Obtener  el  motociclista
seleccionado del JComboBox
        System.out.println("Monto        total:        "        +
orden.getMontoTotal());
        String distanciaConUnidad = orden.getDistancia() + " km";
        String montoConUnidad = "Q. " + orden.getMontoTotal();
        Object[] fila = {
            motociclistaSeleccionado,
            distanciaConUnidad,
            montoConUnidad,
            formatoFecha.format(orden.getFechaCreacion()),
            formatoFecha.format(orden.getFechaLlegadaDerecha())
        };
        modeloHistorial.addRow(fila);
        AppState.serializar();
    }
```

```
}
public void setDistanciaMotociclista(String motociclista, String distancia) {
    if (motociclista.equals("Motociclista 1")) {
        distanciaM1.setText("Distancia: " + distancia + " km");
    } else if (motociclista.equals("Motociclista 2")) {
        distanciaM2.setText("Distancia: " + distancia + " km");
    } else if (motociclista.equals("Motociclista 3")) {
        distanciaM3.setText("Distancia: " + distancia + " km");
    }
}
    private void nuevaOrdenButtonActionPerformed(java.awt.event.ActionEvent evt) {
        reiniciarContadores();
        // Limpia la tabla productosAgregadosJTable si es necesario
        DefaultTableModel modeloProductosAgregados = (DefaultTableModel) productosAgregadosJTable.getModel();
        modeloProductosAgregados.setRowCount(0); // Borra todas las filas
        totalPedido = 0.0; // Reinicia el total del pedido si es necesario
        System.out.println("Reinica el total para una nueva orden");
        totalOrdenLabel.setText("Total del pedido: Q " + totalPedido);
        // Limpia el text field de distancia
```

```
        distanciaTF.setText("");
        // Limpia el combo box
        vehiculoComboBox.setSelectedIndex(0);
    }
```

En este caso, solo se ha colocado el botón para agregar el producto Pizza debido a que los otros cuatro productos tienen el mismo código, lo único que cambia es la variable por el nombre del producto: pollo, papas, gaseosa.

```
    private void agregarPizzaButtonActionPerformed(java.awt.event.ActionEvent evt) {
        contadorPizza++;
        agregarPizzaButton.setText("Agregar (" + contadorPizza + ")");
        double precioPizza = 55;
        Object[] fila = {"Pizza", precioPizza};
        modelo.addRow(fila);
        totalPedido += precioPizza;
        System.out.println("Se suma el precio de la pizza");
        totalOrdenLabel.setText("Total del pedido: Q " + totalPedido);
        Producto pizza = new Producto("Pizza", precioPizza);
        productos.add(pizza);
    }
```

```
private void
confirmarPedidoButtonActionPerformed(java.awt.event.ActionEvent
evt) {
        String distanciaTexto = distanciaTF.getText();
    String motociclistaSeleccionado = (String)
vehiculoComboBox.getSelectedItem();


    if (productos.isEmpty()) {
        JOptionPane.showMessageDialog(this, "No hay productos en
la orden.", "Error", JOptionPane.ERROR_MESSAGE);
        return;
    }


    try {
        int distancia = Integer.parseInt(distanciaTexto);
        if (distancia > 10) {
            JOptionPane.showMessageDialog(this, "La distancia no
puede ser mayor a 10 km", "Error", JOptionPane.ERROR_MESSAGE);
            return;
        }
        int confirmacion = JOptionPane.showConfirmDialog(this,
"¿Desea confirmar la orden?", "Confirmar Orden",
JOptionPane.YES_NO_OPTION);
        System.out.println("Se hace la confirmación");
        if (confirmacion == JOptionPane.YES_OPTION) {
```

```
            // La orden se confirmó
            Orden nuevaOrden = new Orden();
            nuevaOrden.setDistancia(distancia);

nuevaOrden.setMotociclistaSeleccionado(motociclistaSeleccionado);

            Motociclista motociclistaDisponible =
encontrarMotociclistaDisponible(motociclistaSeleccionado);

            if (motociclistaDisponible != null) {

nuevaOrden.setMotociclistaAsignado(motociclistaDisponible);
                motociclistaDisponible.setDisponible(false);
                setDistanciaMotociclista(motociclistaSeleccionado,
String.valueOf(distancia));

                // Agrega los productos a la orden
                for (Producto producto : productos) {
                    nuevaOrden.agregarProducto(producto);
                }

                nuevaOrden.calcularMontoTotal(); // Calcula el
monto total después de agregar los productos
                System.out.println("Se llama a calcular el
monto");
```

```
                Date fechaCreacion = new Date();
                nuevaOrden.setFechaCreacion(fechaCreacion);

                    Calendar calendar = Calendar.getInstance();
                    calendar.setTime(fechaCreacion);
                    calendar.add(Calendar.MINUTE, 27);
                    calendar.add(Calendar.SECOND, 14);
                    Date fechaEntrega = calendar.getTime();
                    nuevaOrden.setFechaEntrega(fechaEntrega);

                    System.out.println("Nueva Orden Agregada:");
                    System.out.println("Motociclista: " +
nuevaOrden.getMotociclistaSeleccionado());
                    System.out.println("Distancia: " +
nuevaOrden.getDistancia() + " km");
                    System.out.println("Monto Total: Q. " +
nuevaOrden.getMontoTotal());
                    System.out.println("Fecha de Creación: " +
nuevaOrden.getFechaCreacion());
                    System.out.println("Fecha de Entrega: " +
nuevaOrden.getFechaLlegadaDerecha());

                    historialPedidos.add(nuevaOrden);
                    actualizarTablaHistorial();
```

```
                    AppState.serializar();
                    System.out.println("Motociclista seleccionado:
" + motociclistaSeleccionado);

                } else {
                    JOptionPane.showMessageDialog(this, "El
motociclista seleccionado no está disponible o no existe.",
"Error", JOptionPane.ERROR_MESSAGE);
                }
            }

        } catch (NumberFormatException e) {
            JOptionPane.showMessageDialog(this, "Error: Por favor,
ingresa una distancia válida", "Error",
JOptionPane.ERROR_MESSAGE);
        }
    }
private void
enviarMotociclista1ButtonActionPerformed(java.awt.event.ActionEven
t evt) {
        if (enviarMotociclista1Habilitado) {
            movimientoImagen1.iniciarMovimiento();
            enviarMotociclista1Habilitado = false;
            enviarMotociclista2Habilitado = true;
            enviarMotociclista3Habilitado = true;
```

```
        }
    }
```

3.2.2.8 NuevoProductoJFrame

Descripción:

La clase NuevoProductoJFrame es una clase de Java que extiende javax.swing.JFrame y se utiliza para crear una ventana de interfaz gráfica que permite al usuario agregar nuevos productos al restaurante.

Atributos:

- restauranteMainFrame (Tipo: RestauranteMainFrame): Una referencia al objeto RestauranteMainFrame que representa la ventana principal de la aplicación.

Constructores:

- NuevoProductoJFrame(RestauranteMainFrame mainFrame): Constructor que inicializa la ventana de agregar productos y recibe una referencia al objeto RestauranteMainFrame para que pueda interactuar con él.

Métodos:

- nombreTFActionPerformed(java.awt.event.ActionEvent evt): Maneja eventos cuando se realiza una acción en el campo de texto del nombre del producto (no implementado en el código proporcionado).
- agregarProductoNuevoButtonActionPerformed(java.awt.event.ActionEvent evt): Maneja eventos cuando se hace clic en el botón "Agregar Producto Nuevo". Toma el nombre y el precio del producto ingresado, valida los datos y agrega el producto a la lista en RestauranteMainFrame.

- regresarButtonActionPerformed(java.awt.event.ActionEvent evt): Maneja eventos cuando se hace clic en el botón "Regresar", cierra la ventana actual y vuelve a hacer visible la ventana principal (RestauranteMainFrame).
- Método main(String args[]): El punto de entrada principal de la aplicación para probar la ventana de agregar productos. Crea una instancia de RestauranteMainFrame y luego crea y muestra la ventana NuevoProductoJFrame con referencia a la ventana principal.

Ejemplos de Uso:

```
private void
agregarProductoNuevoButtonActionPerformed(java.awt.event.Acti
onEvent evt) {
        String nombre = nombreTF.getText().trim();
        String precioProd = precioTF.getText().trim();
        // Verifica si el nombre o el precio están vacíos
        if (nombre.isEmpty() || precioProd.isEmpty()) {
            JOptionPane.showMessageDialog(this, "Debes
completar todos los campos.", "Error",
JOptionPane.ERROR_MESSAGE);
            return; // Sale del método si hay campos vacíos
        }
        try {
            double precio = Double.parseDouble(precioProd);
```

```
            // Verifica si el precio es válido (mayor que
cero)
            if (precio <= 0) {
                JOptionPane.showMessageDialog(this, "El
precio debe ser mayor que cero.", "Error",
JOptionPane.ERROR_MESSAGE);
                return; // Sale del método si el precio no es
válido
            }
            Producto nuevoProducto = new Producto(nombre,
precio);
            // Agrega el producto a la lista en
RestauranteMainFrame
restauranteMainFrame.agregarProducto(nuevoProducto);
// Actualiza la tabla en RestauranteMainFrame
restauranteMainFrame.ActualizarTablaProductos(restauranteMain
Frame.getProducto());
            // Muestra un mensaje de éxito
            JOptionPane.showMessageDialog(this, "Producto
agregado con éxito.", "Éxito",
JOptionPane.INFORMATION_MESSAGE);
            // Limpia los campos para ingresar uno nuevo
            nombreTF.setText("");
            precioTF.setText("");
```

```
        } catch (NumberFormatException e) {
            JOptionPane.showMessageDialog(this, "El precio
debe ser un número válido.", "Error",
JOptionPane.ERROR_MESSAGE);
        }
    }
```

3.2.2.9 FastFoodPanel

Descripción:

La clase FastFoodPanel representa un panel personalizado que muestra una imagen de comida rápida en su interior.

Atributos:

- imagenComida (Tipo: Image): Almacena la imagen de comida rápida que se mostrará en el panel.

Constructores:

- FastFoodPanel(): Constructor de la clase que inicializa el panel y carga la imagen de comida rápida desde un archivo. Si la carga de la imagen falla, se imprime un mensaje de error.

Métodos:

- paintComponent(Graphics g): Método sobrescrito de javax.swing.JPanel que se encarga de dibujar la imagen de comida rápida en el panel. Redimensiona la imagen para que se ajuste al tamaño del panel y la dibuja en la ubicación (0, 0) del panel.

- initComponents(): Método generado automáticamente por el editor de formularios que inicializa los componentes del panel. En este caso, no contiene ningún componente adicional aparte del fondo del panel.

3.2.2.10 GaseosaPanel

Descripción:

La clase GaseosaPanel representa un panel personalizado que muestra una imagen de una lata de gaseosa en su interior.

Atributos:

- imagenGaseosa (Tipo: Image): Almacena la imagen de una lata de gaseosa que se mostrará en el panel.

Métodos:

- paintComponent(Graphics g): Método sobrescrito de javax.swing.JPanel que se encarga de dibujar la imagen de la lata de gaseosa en el panel. Redimensiona la imagen para que se ajuste al tamaño del panel y la dibuja en la ubicación (0, 0) del panel.
- initComponents(): Método generado automáticamente por el editor de formularios que inicializa los componentes del panel. En este caso, no contiene ningún componente adicional aparte del fondo del panel.

3.2.2.11 PapasPanel

Descripción:

La clase PapasPanel representa un panel personalizado que muestra una imagen de papas fritas en su interior.

Atributos:

- imagenPapas (Tipo: Image): Almacena la imagen de papas fritas que se mostrará en el panel.

Constructores:

- PapasPanel(): Constructor de la clase que inicializa el panel y carga la imagen de las papas fritas desde un archivo. Si la carga de la imagen falla, se imprime un mensaje de error.

Métodos:

- paintComponent(Graphics g): Método sobrescrito de javax.swing.JPanel que se encarga de dibujar la imagen de las papas fritas en el panel. Redimensiona la imagen para que se ajuste al tamaño del panel y la dibuja en la ubicación (0, 0) del panel.
- initComponents(): Método generado automáticamente por el editor de formularios que inicializa los componentes del panel. En este caso, no contiene ningún componente adicional aparte del fondo del panel.

3.2.2.12 PizzaPanel

Descripción:

La clase PizzaPanel representa un panel personalizado que muestra una imagen de pizza en su interior.

Atributos:

- imagenPizza (Tipo: Image): Almacena la imagen de pizza que se mostrará en el panel.

Constructores:

- PizzaPanel(): Constructor de la clase que inicializa el panel y carga la imagen de pizza desde un archivo. Si la carga de la imagen falla, se imprime un mensaje de error.

Métodos:

- paintComponent(Graphics g): Método sobrescrito de javax.swing.JPanel que se encarga de dibujar la imagen de pizza en el panel. Redimensiona la imagen para que se ajuste al tamaño del panel y la dibuja en la ubicación (0, 0) del panel.
- initComponents(): Método generado automáticamente por el editor de formularios que inicializa los componentes del panel. En este caso, no contiene ningún componente adicional aparte del fondo del panel.

3.2.2.13 PolloPanel

Descripción:

La clase PolloPanel representa un panel personalizado que muestra una imagen de pollo frito en su interior.

Atributos:

- imagenPollo (Tipo: Image): Almacena la imagen de pollo frito que se mostrará en el panel.

Constructores:

- PolloPanel(): Constructor de la clase que inicializa el panel y carga la imagen de pollo frito desde un archivo. Si la carga de la imagen falla, se imprime un mensaje de error.

Métodos:

- paintComponent(Graphics g): Método sobrescrito de javax.swing.JPanel que se encarga de dibujar la imagen de pollo frito en el panel. Redimensiona la imagen para que se ajuste al tamaño del panel y la dibuja en la ubicación (0, 0) del panel.
- initComponents(): Método generado automáticamente por el editor de formularios que inicializa los componentes del panel. En este caso, no contiene ningún componente adicional aparte del fondo del panel.

3.3 Funcionalidades de la Aplicación

3.3.1 Funcionalidades

- Agregar nuevos productos: En la ventana principal se puede seleccionar el botón para agregar manualmente un nuevo producto al sistema con su respectivo nombre y precio.

- Productos por defecto: El sistema cuenta con cuatro productos por defecto dado que son los más populares o los más vendidos por lo que no hay necesidad de agregarlos manualmente.
- Orden: Se puede observar un listado con los productos que se han seleccionado incluyendo su nombre, precio y el monto total del pedido a pagar.
- Distancia: El usuario debe ingresar la distancia a la cuál será enviado el pedido tomando en cuenta que esta no puede ser mayor a 10 km.
- Motociclista: Se debe seleccionar a uno de los motociclistas del personal quien será el encargado del delivery del pedido, considerando que solo se puede enviar a un motociclista a la vez o todos al mismo tiempo.
- Control de Delivery: En esta sección, se gestiona el kilometraje del repartidor y su ubicación gráficamente al haberlo enviado con el pedido.
- Historial: Funciona como un listado de los pedidos realizados con su información como: el motociclista enviado, la distancia en kilómetros, el monto total del pedido en quetzales, la fecha y hora de creación del pedido y la fecha y hora de entrega del mismo.

3.4 Interfaz de Usuario

Se utilizó la librería Swing para crear la interfaz del usuario, incluyendo la configuración de ventanas y elementos de interfaz de usuario como:

- Botones: los cuales el usuario debe oprimir para acceder a sus funcionalidades.
- Combo Box: contienen opciones que el usuario debe seleccionar o escoger.
- Text Field o Campos de texto: donde el usuario ingresa la información.
- Panel: permite utilizar múltiples layouts y que los componentes sobre la ventana de visualización puedan modificarse con mucha flexibilidad.
- Tabbed Pane: permite dividir especialmente la ventana en pestañas.
- Tablas las cuales son utilizadas para mostrar la información de los profesores, cursos y estudiantes.

3.5 Ejecución

3.5.1 Archivo JAR Ejecutable

- Asegurarse de que Java esté instalado en el sistema antes de intentar ejecutar la aplicación desde un archivo JAR.
- Descargar el archivo JAR de la aplicación: asegurarse de haber descargado el archivo JAR para proceder a ejecutarlo.
- Navegar en el directorio de la aplicación y encontrar la carpeta en donde está almacenado.
- Ejecutar la aplicación
- Interactuar con la aplicación: la aplicación debería comenzar a ejecutarse, por lo tanto, se debe de tener acceso a las funcionalidades de la misma.
- Cerrar la aplicación

3.6 Áreas de Mejoras Futuras

Se considerará en el futuro mejorar la opción para gestionar que la velocidad del repartidor para tener un mejor control y visibilidad del mismo en su trayectoria al igual que la persistencia de datos de las ordenes que se realizan, mejorar su interfaz y las funcionalidades en general.