UNIVERSIDAD DE SAN CARLOS DE GUATEMALA
FACULTAD DE INGENIERÍA
ESTRUCTURA DE DATOS
CATEDRÁTICO: ING. WILLIAM ESTUARDO ESCOBAR ARGUETA
TUTOR ACADÉMICO: JOSUÉ RODOLFO MORALES CASTILLO

MANUAL TÉCNICO

NOMBRE COMPLETO
CARNÉ: 202300848
SECCIÓN: B

GUATEMALA, 1 DE ABRIL DEL 2,024

ÍNDICE

Tabla de contenido

INTRODUCCIÓN

Este proyecto ofrece una aplicación para ayudar a los usuarios a gestionar y monitorear sus viajes. Diseñado para ser intuitivo y fácil de usar, el sistema permite a los usuarios agregar, seguir, y revisar viajes mediante una interfaz gráfica amigable. Los usuarios pueden cargar datos de rutas desde archivos CSV, visualizar un historial detallado de sus viajes, incluyendo información sobre los vehículos usados y el combustible consumido.

La aplicación está construida con varias clases Java, cada una enfocada en una tarea específica, como manejar la interfaz de usuario, gestionar los datos de los viajes, y guardar/cargar el estado de la aplicación para asegurar una experiencia de usuario coherente. Las clases trabajan juntas para ofrecer funcionalidades como la planificación de viajes, el seguimiento en tiempo real, y análisis del historial de viajes.

OBJETIVOS

1. GENERAL

1.1. Proporcionar una guía detallada sobre el uso y la administración del Sistema de Gestión de Viajes

2. ESPECÍFICOS

2.1. Dar a conocer los metodos mas importantes y cual es su uso

2.2. Detallar el objetivo.

ALCANCES DEL SISTEMA

El objetivo de este manual es proporcionar una guía detallada sobre el uso y la administración del Sistema de Gestión de Viajes. Pretende ser una herramienta de consulta para usuarios finales y desarrolladores, facilitando la comprensión del

funcionamiento del sistema, la navegación por su interfaz, y el aprovechamiento óptimo de sus características y funcionalidades. Este manual incluye instrucciones paso a paso, explicaciones de las distintas secciones del sistema, y consejos útiles para resolver problemas comunes, asegurando así que los usuarios puedan gestionar sus viajes de manera eficiente y efectiva.

ESPECIFICACIÓN TÉCNICA

- **REQUISITOS DE HARDWARE**

Para garantizar el correcto desarrollo y mantenimiento de la aplicación, se recomienda que el programador cuente con las siguientes especificaciones mínimas de hardware:

  - Procesador: Intel Core i5 o superior / AMD equivalente.
  - Memoria RAM: 8 GB como mínimo, recomendado 16 GB para un mejor rendimiento.
  - Espacio en Disco Duro: Al menos 1 GB de espacio libre para la instalación de herramientas de desarrollo y almacenamiento de proyectos.

- **REQUISITOS DE SOFTWARE**

El programador necesitará los siguientes softwares para trabajar en la aplicación y facilitar el desarrollo futuro:

- Sistema Operativo: Windows 10 o superior, macOS Mojave o superior, o una distribución reciente de Linux (Ubuntu 20.04 LTS recomendado).
- Entorno de Desarrollo Integrado (IDE): IntelliJ IDEA, Eclipse, o NetBeans para el desarrollo en Java. La elección dependerá de las preferencias personales y las necesidades específicas del proyecto.
- JDK: Java Development Kit (JDK) versión 11 o superior. Es esencial para el desarrollo de aplicaciones Java.

- Control de Versiones: Git, con repositorios en plataformas como GitHub o GitLab, para el manejo de versiones del código fuente.
- Herramientas de Construcción y Gestión de Dependencias: Maven o Gradle, necesarios para la construcción de proyectos y gestión de bibliotecas.

DESCRIPCIÓN DE LA SOLUCIÓN

Para abordar los requerimientos, comenzamos por analizar detenidamente las especificaciones proporcionadas, identificando las principales funcionalidades y objetivos del sistema. Entendimos que el núcleo del proyecto giraba en torno a la gestión de viajes y rutas, proporcionando una interfaz interactiva para la creación, visualización y gestión de estos elementos.

Con este entendimiento, procedimos a dividir el proyecto en componentes claramente definidos:

1. Interfaz de Usuario (UI): Diseñamos una interfaz gráfica intuitiva utilizando Swing, permitiendo a los usuarios interactuar de manera efectiva con la aplicación. Nos enfocamos en hacer que la navegación fuera sencilla y que las acciones más importantes estuvieran fácilmente accesibles.
2. Gestión de Datos: Implementamos un sistema robusto para el manejo de la información relacionada con los viajes y rutas. Esto incluyó la creación de clases para representar estos datos, así como la serialización de objetos para su almacenamiento y recuperación.
3. Lógica de Negocio: Desarrollamos la lógica necesaria para conectar la interfaz de usuario con el sistema de gestión de datos, asegurando que las acciones realizadas en la UI se reflejaran correctamente en el modelo de datos del sistema.

Para cada uno de estos componentes, adoptamos un enfoque iterativo, desarrollando y refinando funcionalidades a medida que avanzábamos. Este método nos permitió adaptarnos a nuevos requerimientos y hacer ajustes basados en pruebas y feedback temprano.

Al enfrentar desafíos técnicos, como la serialización de objetos y la gestión de hilos para procesos en segundo plano, recurrimos a la documentación oficial y recursos de la comunidad de desarrolladores para encontrar soluciones óptimas y aplicar las mejores prácticas.

Finalmente, mantuvimos un compromiso constante con la calidad del código y la documentación, lo cual facilitará futuras extensiones o modificaciones del sistema por parte de otros desarrolladores. La solución resultante es un sistema flexible y fácil de usar que cumple con los requerimientos establecidos y ofrece una base sólida para la expansión futura.

LÓGICA DEL PROGRAMA

❖ **Travels:**

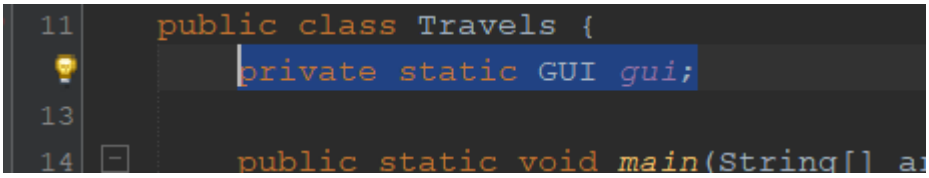
```
package travels;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.ArrayList;
import javax.swing.SwingUtilities;
```

➢ **Librerías**

- java.io.*: Proporciona las clases necesarias para operaciones de entrada y salida a través de flujos de datos, serialización y el sistema de archivos. Se usa para guardar y cargar el estado de la aplicación a un archivo.
- java.util.ArrayList: Permite crear arrays dinámicos que pueden aumentar o disminuir su tamaño automáticamente. Se utiliza para gestionar listas de datos, como registros de viajes y estados de viajes.
- javax.swing.SwingUtilities: Proporciona métodos útiles para trabajar con la biblioteca gráfica Swing. En este caso, se utiliza para asegurar que la interfaz gráfica de usuario (GUI) se inicie en el Event Dispatch Thread (EDT), que es el hilo apropiado para las actualizaciones de la interfaz gráfica.

Variables Globales de la clase _(gui)



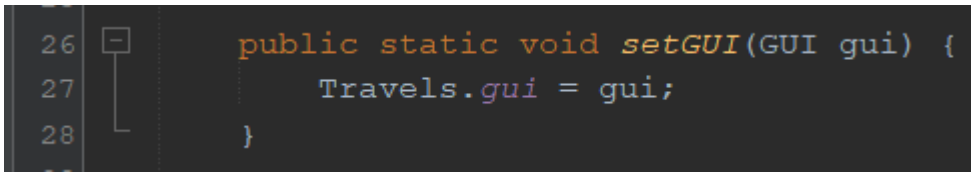
private static GUI gui: Variable estática que mantiene una referencia a la interfaz gráfica de usuario principal de la aplicación. Se utiliza para interactuar con la GUI desde diferentes partes del programa.

➢ **Función Main**

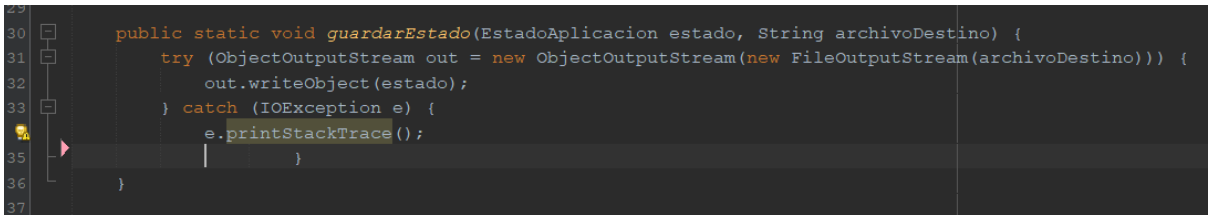
El método main es el punto de entrada de la aplicación. Utiliza SwingUtilities.invokeLater para asegurar que la creación y actualización de la interfaz gráfica de usuario (GUI) se realice en el EDT. Dentro de este método, se intenta cargar un estado previamente guardado de la aplicación desde un archivo. Si el archivo no existe o hay un error al cargarlo, se inicializa un nuevo estado con valores predeterminados. Finalmente, se crea y muestra la GUI principal de la aplicación.

➢ **Métodos y Funciones utilizadas**

- public static void setGUI(GUI gui): Establece la referencia a la GUI principal. Permite la comunicación y manipulación de la GUI desde otras partes del código.

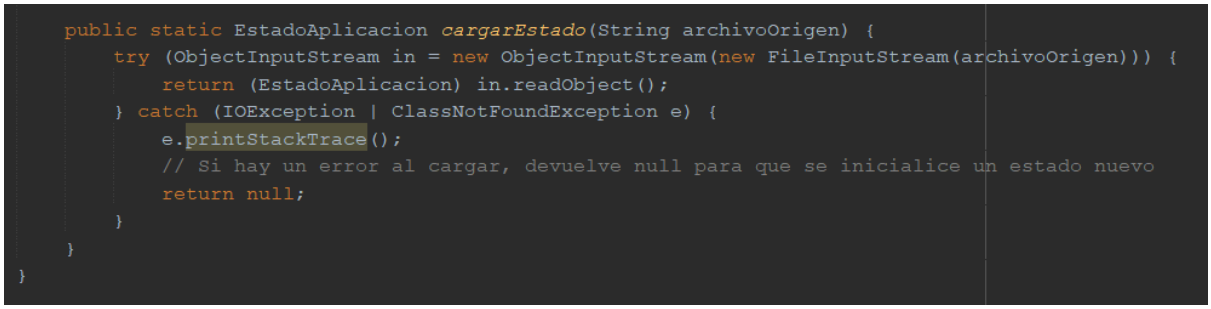


- public static void guardarEstado(EstadoAplicacion estado, String archivoDestino): Serializa y guarda el estado actual de la aplicación (incluidos los registros de viajes, el historial, etc.) en un archivo especificado. Maneja las excepciones relacionadas con operaciones de entrada/salida.



```
    public static void guardarEstado(EstadoAplicacion estado, String archivoDestino) {
        try (ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(archivoDestino))) {
            out.writeObject(estado);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

- public static EstadoAplicacion cargarEstado(String archivoOrigen): Intenta cargar y deserializar el estado de la aplicación desde un archivo especificado. Maneja las excepciones y devuelve null si no puede cargar el estado, lo que indica que se debe inicializar un nuevo estado.



```
    public static EstadoAplicacion cargarEstado(String archivoOrigen) {
        try (ObjectInputStream in = new ObjectInputStream(new FileInputStream(archivoOrigen))) {
            return (EstadoAplicacion) in.readObject();
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
            // Si hay un error al cargar, devuelve null para que se inicialice un estado nuevo
            return null;
        }
    }
}
```

❖ TABLE RUTES

➢ Librerías

```
1   package travels;
2
3   import javax.swing.*;
4   import javax.swing.table.DefaultTableModel;
5   import java.awt.*;
6   import java.awt.event.*;
7   import java.io.*;
8   import java.util.ArrayList;
9   import javax.swing.table.JTableHeader;
10  import javax.swing.table.TableCellRenderer;
11  import javax.swing.table.TableColumn;
12  import javax.swing.table.TableColumnModel;
13
```

- javax.swing.*: Proporciona clases e interfaces para la creación de componentes de la interfaz gráfica de usuario. Se utiliza para manejar la tabla de rutas, botones, diálogos, entre otros componentes gráficos.
- java.awt.*: Contiene todas las clases para crear interfaces gráficas de usuario e imágenes. Se usa para detalles de layout y configuración de componentes.
- java.io.*: Proporciona las clases necesarias para operaciones de entrada y salida. Se utiliza para leer y escribir archivos CSV que contienen datos de rutas.
- java.util.ArrayList: Se usa para manejar listas dinámicas de registros de rutas y para mantener las listas de inicios y finales de rutas.

➢ Variables Globales

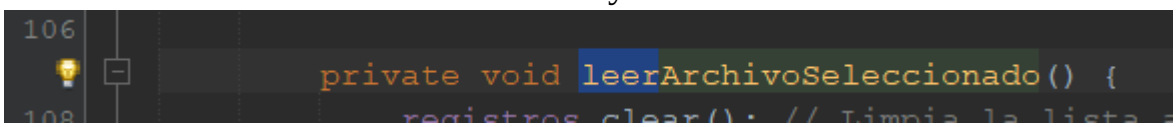
```
JTable rutas;
DefaultTableModel modeloTabla = new DefaultTableModel(new Object[]{"ID", "INICIO", "FIN", "Distancia"}, 0);
ArrayList<RegistroCSV> registros = new ArrayList<>();
private static File selectedFile;
public ArrayList<String> inicios = new ArrayList<>();
public ArrayList<String> finals = new ArrayList<>();
```

- JTable rutas: Muestra los registros de rutas en una interfaz gráfica tabular.
- DefaultTableModel modeloTabla: Modelo para manejar los datos mostrados en rutas.
- ArrayList<RegistroCSV> registros: Almacena los registros de las rutas cargadas desde un archivo CSV o añadidas durante la ejecución.
- File selectedFile: Mantiene referencia al archivo CSV seleccionado por el usuario.
- ArrayList<String> inicios, ArrayList<String> finals: Listas para almacenar puntos de inicio y fin de las rutas, respectivamente.

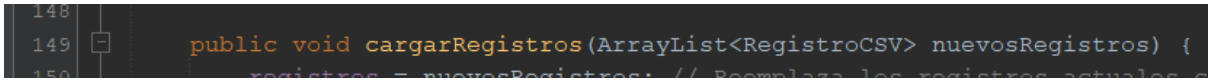
➢ **Métodos y Funciones utilizadas**

- public void setTripsPanel(TripsPanel tripsPanel): Método placeholder para establecer una referencia al panel de viajes, si se necesita en el futuro.

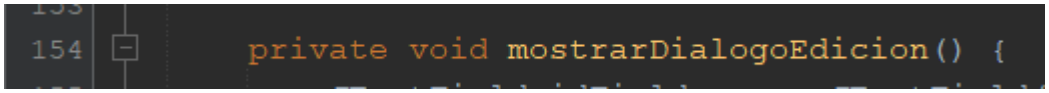
- private void leerArchivoSeleccionado(): Lee un archivo CSV seleccionado por el usuario, extrayendo datos de rutas y actualizando la tabla y listas relacionadas.



- public void cargarRegistros(ArrayList<RegistroCSV> nuevosRegistros): Carga nuevos registros en la tabla, reemplazando los existentes.



- private void mostrarDialogoEdicion(), private void editarDistancia(String id, String nuevaDistancia): Muestra un diálogo para editar la distancia de un registro existente y actualiza el registro correspondiente.



- private void escribirRegistrosACSV(): Guarda los registros actuales en el archivo CSV seleccionado.

```
183     private void escribirRegistrosACSV() {
```

- private void agregarRuta(String inicio, String fin, String distancia): Añade un nuevo registro de ruta a la lista y actualiza la tabla.

```
private void agregarRuta(String inicio, String fin, String distancia) {
```

- private void eliminarRuta(String id): Elimina un registro de ruta basado en su ID.

```
238     private void eliminarRuta(String id) {
```

- public void ajustarAnchoColumnas(JTable tabla): Ajusta el ancho de las columnas de la tabla para adaptarse al contenido.

```
256 }
    public void ajustarAnchoColumnas(JTable tabla) {
258     // Asegura que el layout de la tabla esté actualizad
259     tabla.setAutoResizeMode(JTable.AUTO_RESIZE_ALL_COLUM
```

-

❖ **TRIPS** **PANEL**

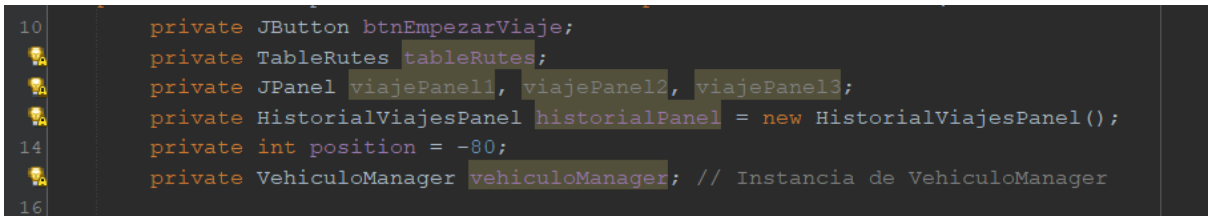
```
3  import javax.swing.*;
4  import java.awt.*;
5  import java.awt.event.ActionEvent;
6  import java.io.Serializable;
7  import java.util.ArrayList;
```

➢ **Librerías**

- javax.swing.*: Usada para construir la interfaz gráfica del panel de viajes. Proporciona componentes como botones (JButton), diálogos (JDialog), paneles (JPanel), y comboboxes (JComboBox).
- java.awt.*: Proporciona clases para la gestión de eventos (ActionEvent) y la configuración de la interfaz gráfica (GridBagConstraints, Insets).
- java.io.Serializable: Se implementa para permitir la serialización de TripsPanel, lo cual es útil para guardar el estado de la aplicación si es necesario.

- java.util.ArrayList: Se utiliza para manejar listas de objetos, particularmente para administrar vehículos disponibles.

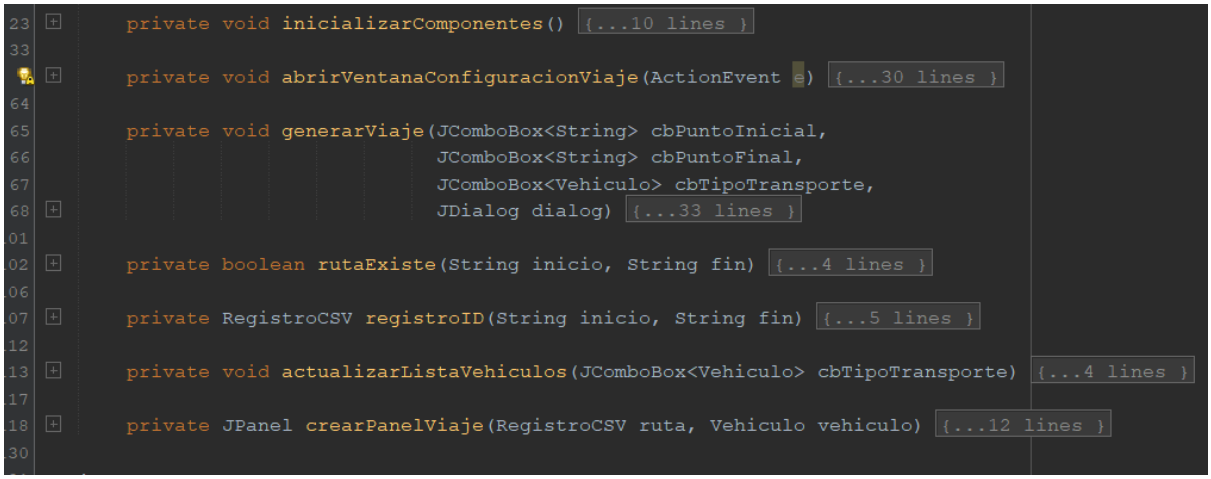
➢ **Variables Globales de la clase _(El nombre de su clase actual)**



❖ JButton btnEmpezarViaje: Botón que inicia la configuración de un nuevo viaje.

- TableRutes tableRutes: Referencia a la instancia de TableRutes para acceder a la información de rutas disponibles.
- JPanel viajePanel1, viajePanel2, viajePanel3: Paneles para mostrar los viajes en curso (no se utilizan explícitamente en el fragmento proporcionado, pero podrían ser parte de una expansión).
- HistorialViajesPanel historialPanel: Panel para registrar y mostrar el historial de viajes completados.
- int position: Controla la posición de los paneles de viaje en curso dentro del TripsPanel.
- VehiculoManager vehiculoManager: Gestiona la información y disponibilidad de vehículos.

➢ **Métodos y Funciones utilizadas**



- inicializarComponentes(): Configura los componentes iniciales del panel, incluyendo el botón para empezar un nuevo viaje.
- abrirVentanaConfiguracionViaje(ActionEvent e): Despliega una ventana de diálogo para configurar los detalles de un nuevo viaje, permitiendo al usuario seleccionar puntos inicial y final, así como el tipo de vehículo.
- generarViaje(...): Valida si existe una ruta entre los puntos seleccionados y, de ser así, inicia un nuevo hilo para simular el viaje.
- rutaExiste(String inicio, String fin), registroID(String inicio, String fin): Funciones de ayuda para verificar la existencia de una ruta y para obtener el registro correspondiente.
- actualizarListaVehiculos(JComboBox<Vehiculo> cbTipoTransporte): Actualiza la lista de vehículos disponibles en el combobox de vehículos.
- crearPanelViaje(RegistroCSV ruta, Vehiculo vehiculo): Crea un panel para visualizar un viaje en curso.

❖ GUI

```
2
4   import java.awt.Color;
5   import javax.swing.*;
6   import java.awt.event.WindowAdapter;
    import java.awt.event.WindowEvent;
    import java.util.ArrayList;
```

➢ Librerías

- javax.swing.*: Para construir y gestionar los componentes de la interfaz gráfica de usuario, como frames (JFrame), paneles (JPanel), y pestañas (JTabbedPane).
- java.awt.Color: Para definir colores que se utilizan en la interfaz gráfica.
- java.awt.event.*: Para manejar eventos como el cierre de la ventana (WindowEvent).
- java.util.ArrayList: Se usa para manejar listas de registros y viajes.

➢ Variables Globales de la clase

```
9   public class GUI extends JFrame {
10      private TableRutes tableRutes;
11      private HistorialViajesPanel historial;
```

- TableRutes tableRutes: Gestiona la interfaz y lógica relacionada con las rutas de viaje.
- HistorialViajesPanel historial: Panel que muestra el historial de viajes realizados.

➢ Métodos y Funciones utilizadas

```
12
13  public GUI(EstadoAplicacion estadoCargado) {...40 lines }
53  public void agregarNuevoViaje(RegistroCSV nuevoRegistro, ViajeRealizado.Viaje nuevoViaje) {...14 lines }
67  }
```

- GUI(EstadoAplicacion estadoCargado): Constructor que inicializa la ventana principal de la aplicación, configura sus

componentes y carga el estado de la aplicación si está disponible.

- agregarNuevoViaje(RegistroCSV nuevoRegistro, ViajeRealizado.Viaje nuevoViaje): Agrega un nuevo viaje a los registros y al historial, luego guarda el nuevo estado de la aplicación.

❖ HistorialViajesPanel

```
3  import javax.swing.*;
4  import java.awt.*;
5  import javax.swing.table.DefaultTableModel;
6  import java.util.ArrayList;
   import java.util.Date;
8  import java.io.Serializable;
9  import travels.ViajeRealizado.Viaje;
```

➢ Librerías

- javax.swing.*: Utilizada para crear componentes de la interfaz gráfica como paneles (JPanel), tablas (JTable), y el modelo de tabla (DefaultTableModel).
- java.awt.*: Usada para manejar componentes y propiedades de diseño como colores (Color) y dimensiones (Dimension).
- java.util.ArrayList: Necesaria para gestionar la lista de viajes (ArrayList<Viaje>).
- java.util.Date: Usada para representar fechas y horas de inicio y fin de los viajes.
- java.io.Serializable: Implementada para permitir la serialización de HistorialViajesPanel, facilitando la persistencia de datos.

➢ **Variables Globales de la clase**

```
public class HistorialViajesPanel extends JPanel implements Serializable {
    private JTable historialTable;
    private DefaultTableModel tableModel;
    private ArrayList<Viaje> listaViajes;
```

- JTable historialTable: La tabla que muestra el historial de viajes.
- DefaultTableModel tableModel: El modelo de datos para historialTable.
- ArrayList<Viaje> listaViajes: Una lista que almacena los viajes realizados.

➢ **Métodos y Funciones utilizadas**

```
16    public HistorialViajesPanel() {...6 lines }
22    private void initUIComponents() {...18 lines }
40    public void agregarViajeYActualizarTabla(Viaje viaje) {...7 lines }
47    public void cargarHistorial(ArrayList<Viaje> historialViajes) {...7 lines }
54    public void actualizarTabla() {...33 lines }
87    public ArrayList<Viaje> getListaViajes() {...3 lines }
90    }
```

- HistorialViajesPanel(): Constructor que inicializa la interfaz del panel de historial de viajes, configurando la tabla y cargando cualquier dato previamente existente.
- initUIComponents(): Configura los componentes de UI del panel, incluyendo la tabla y su modelo.
- agregarViajeYActualizarTabla(Viaje viaje): Añade un nuevo viaje al historial y actualiza la tabla para reflejar el cambio.
- cargarHistorial(ArrayList<Viaje> historialViajes): Carga un conjunto de viajes al historial desde una fuente externa, por ejemplo, al iniciar la aplicación.
- actualizarTabla(): Refresca los datos mostrados en la tabla para que coincidan con el estado actual de listaViajes.

❖ **ViajeEnCurso**

```
3  import javax.swing.*;
4  import java.awt.*;
5  import java.awt.image.BufferedImage;
6  import java.io.Serializable;
7  import java.util.concurrent.atomic.AtomicBoolean;
8  import java.text.SimpleDateFormat;
9  import java.util.Date;
   import java.util.UUID;
```

➢ **Librerías**

- javax.swing.*: Para la creación de componentes de interfaz gráfica como JPanel, JLabel, JButton, y ImageIcon.
- java.awt.*: Para manejo de colores (Color), dimensiones (Dimension), imágenes (BufferedImage), y dibujo en el panel (Graphics).
- java.io.Serializable: Permite que ViajeEnCurso pueda ser serializado, facilitando su almacenamiento o transmisión.

- java.util.concurrent.atomic.AtomicBoolean: Utilizado para controlar el estado de animación de manera segura en entornos multihilo.
- java.text.SimpleDateFormat: Para formatear fechas y horas de inicio y fin de los viajes.
- java.util.Date: Representa fechas y horas específicas, utilizadas para marcar el inicio y el fin del viaje.
- java.util.UUID: Genera identificadores únicos, posiblemente utilizado para identificar viajes de manera única.

➢ **Variables Globales de la clase**

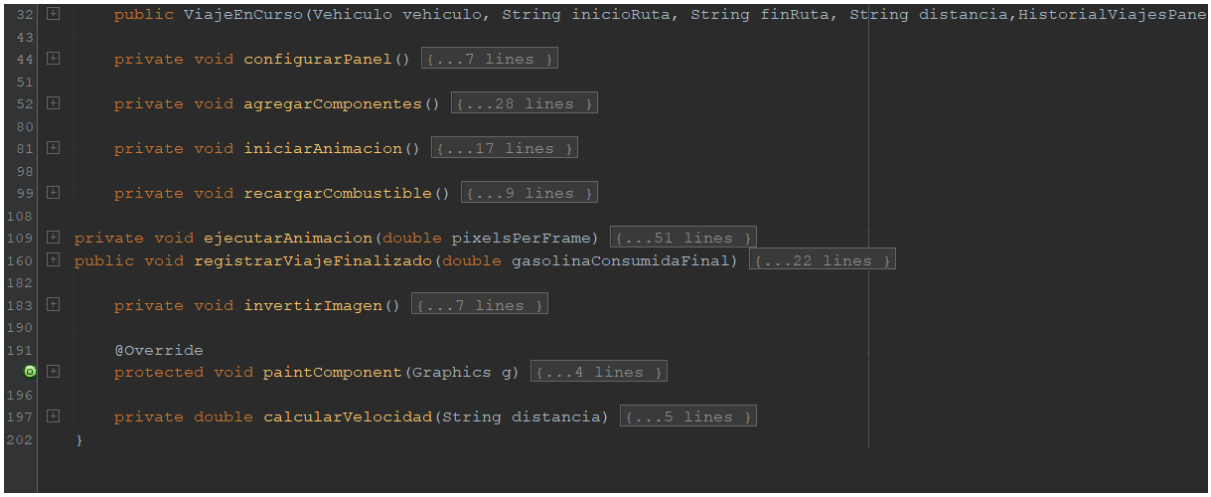
```
Date fechaHoraInicio = new Date();
Date fechaHoraFin = new Date();
SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
private HistorialViajesPanel historialPanel;
private ViajeRealizado.Viaje viaje;
private ImageIcon imagenVehiculo;
private JLabel lblCombustible;
private double gasolinaConsumida = 0;
private volatile int xPosition = 620; // Posición inicial
private final String inicioRuta, finRuta, tipoVehiculo, distanciaTotal;
private Double velocidad;

private AtomicBoolean enMovimiento = new AtomicBoolean(false);
private int animacionDireccion = -1; // Define la dirección inicial de la animación para ir hacia la izquierda
private JButton btnRecargarCombustible;
private Vehiculo vehiculo;
private boolean primeraVezIniciado = false;
```

- ImageIcon imagenVehiculo: Representa visualmente el vehículo.
- JLabel lblCombustible: Muestra la cantidad de combustible consumido.
- AtomicBoolean enMovimiento: Controla si la animación del viaje está activa.
- int xPosition: La posición horizontal del vehículo en la animación.
- String inicioRuta, finRuta, tipoVehiculo, distanciaTotal: Datos del viaje.
- Double velocidad: Velocidad de la animación basada en la distancia del viaje.

- JButton btnRecargarCombustible: Botón para recargar combustible del vehículo.

➢ Métodos y Funciones utilizadas

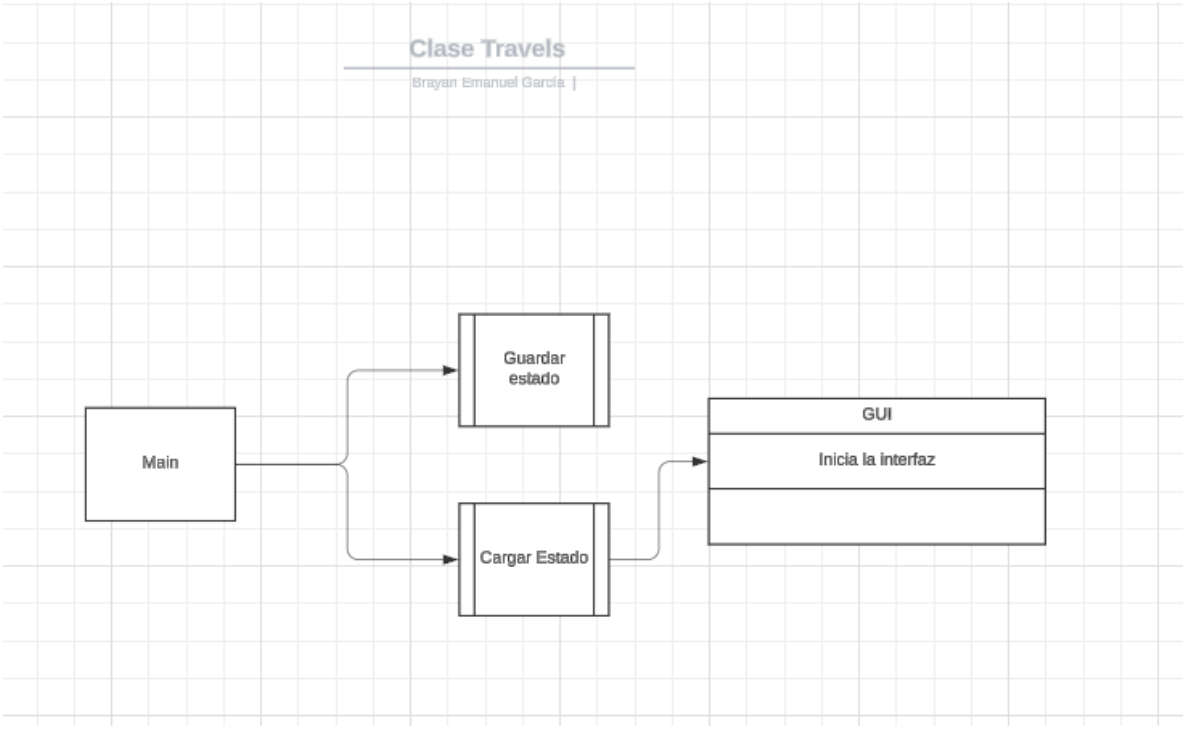


- ViajeEnCurso(): Constructor que inicializa el panel del viaje en curso, configurando la interfaz y almacenando los datos del viaje.
- configurarPanel(): Prepara la interfaz del panel, estableciendo tamaño, color de fondo, y disposición de los componentes.
- agregarComponentes(): Añade y configura los componentes UI como etiquetas y botones, incluyendo acciones para iniciar el viaje y recargar combustible.
- iniciarAnimacion(): Inicia o reinicia la animación del viaje, controlando el consumo de combustible y la posición del vehículo.
- recargarCombustible(): Recarga el combustible del vehículo y, si es necesario, reinicia la animación.
- ejecutarAnimacion(double): Ejecuta la lógica de animación, actualizando la posición del vehículo y el combustible consumido en tiempo real.
- registrarViajeFinalizado(double): Registra el viaje como completado, actualizando el historial de viajes con los detalles finales.
- invertirImagen(): Invierte la imagen del vehículo para cambiar su dirección de movimiento en la animación.

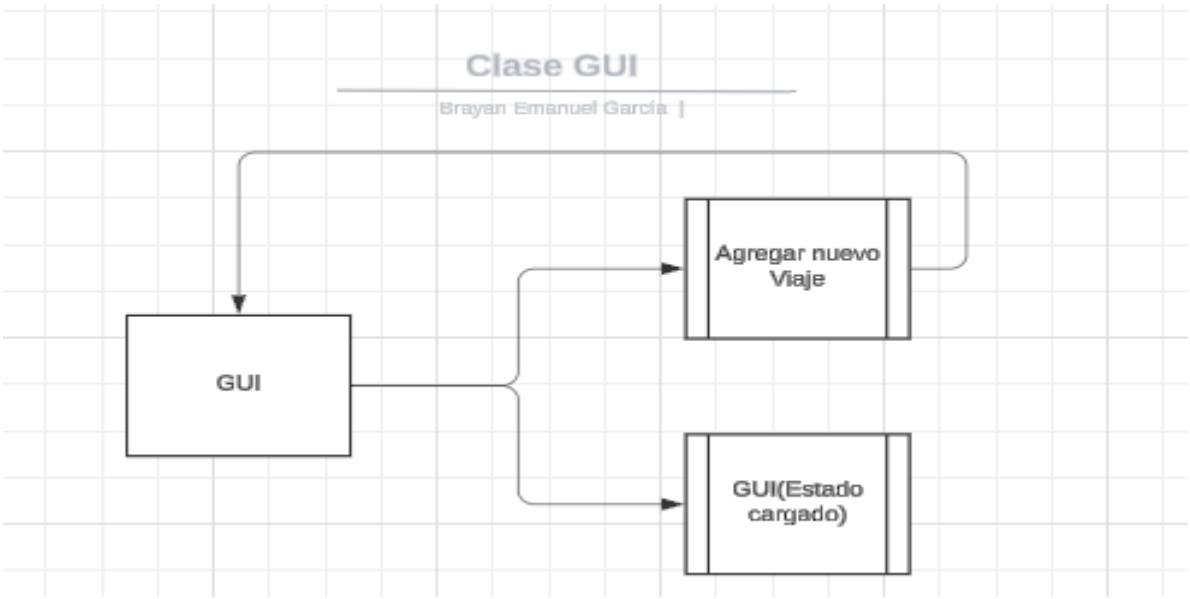
- calcularVelocidad(String): Calcula la velocidad de la animación basada en la distancia total del viaje.

Diagramas de clases

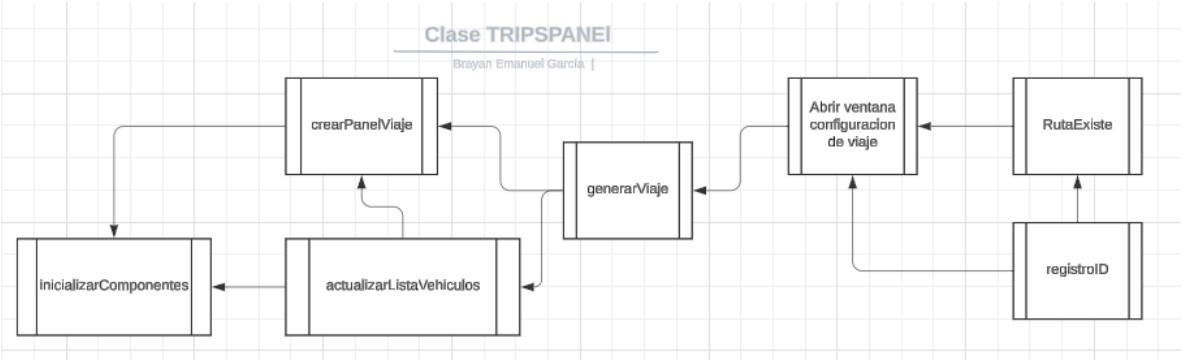
Travels



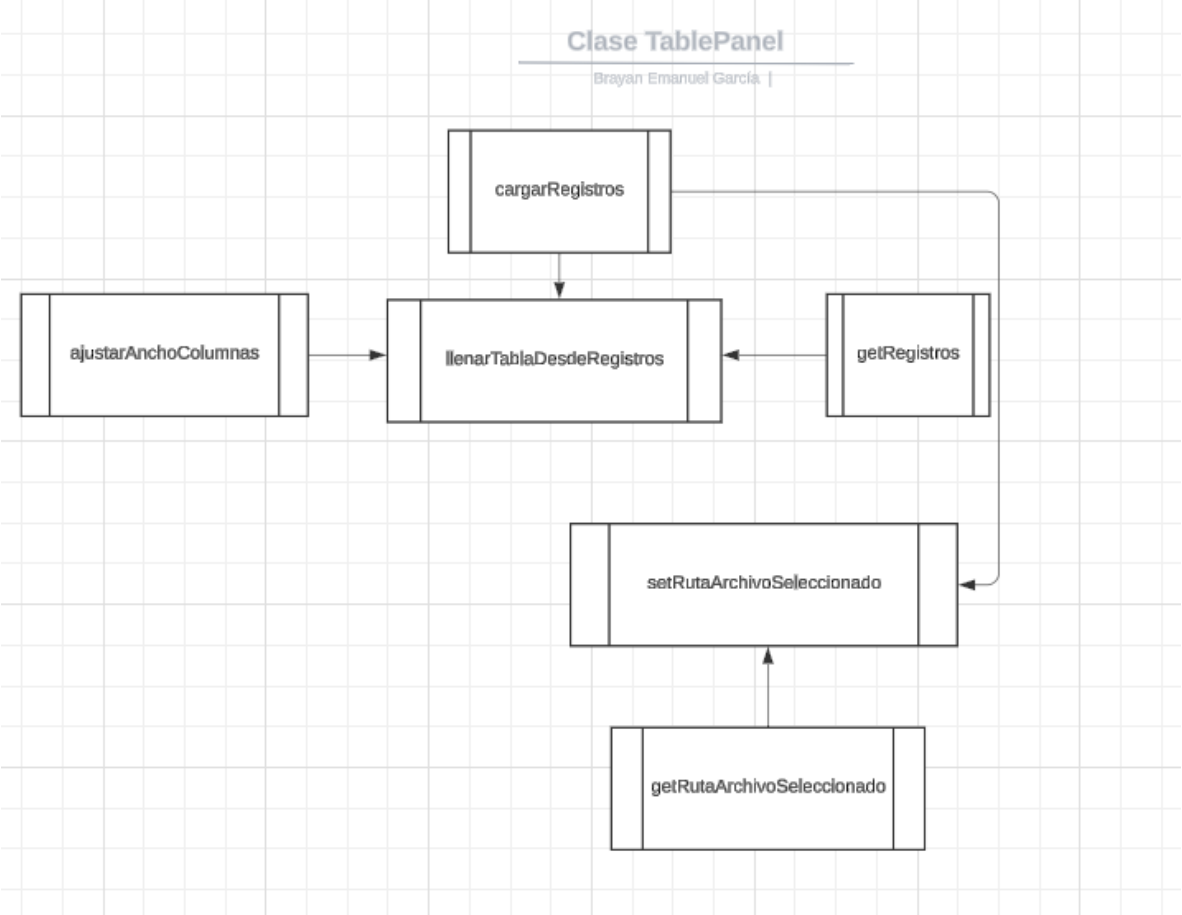
GUI



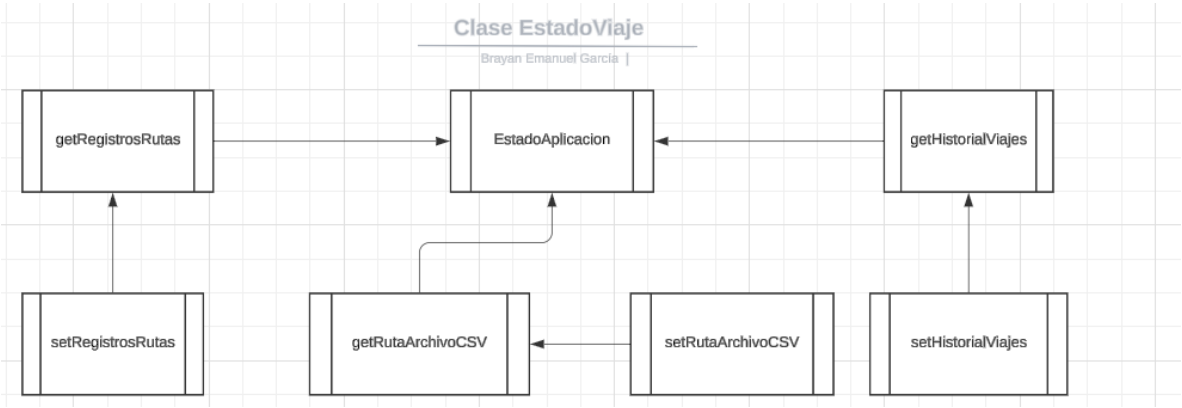
Trips Panel



TablePanel



Estado Viajes



Viaje en curso

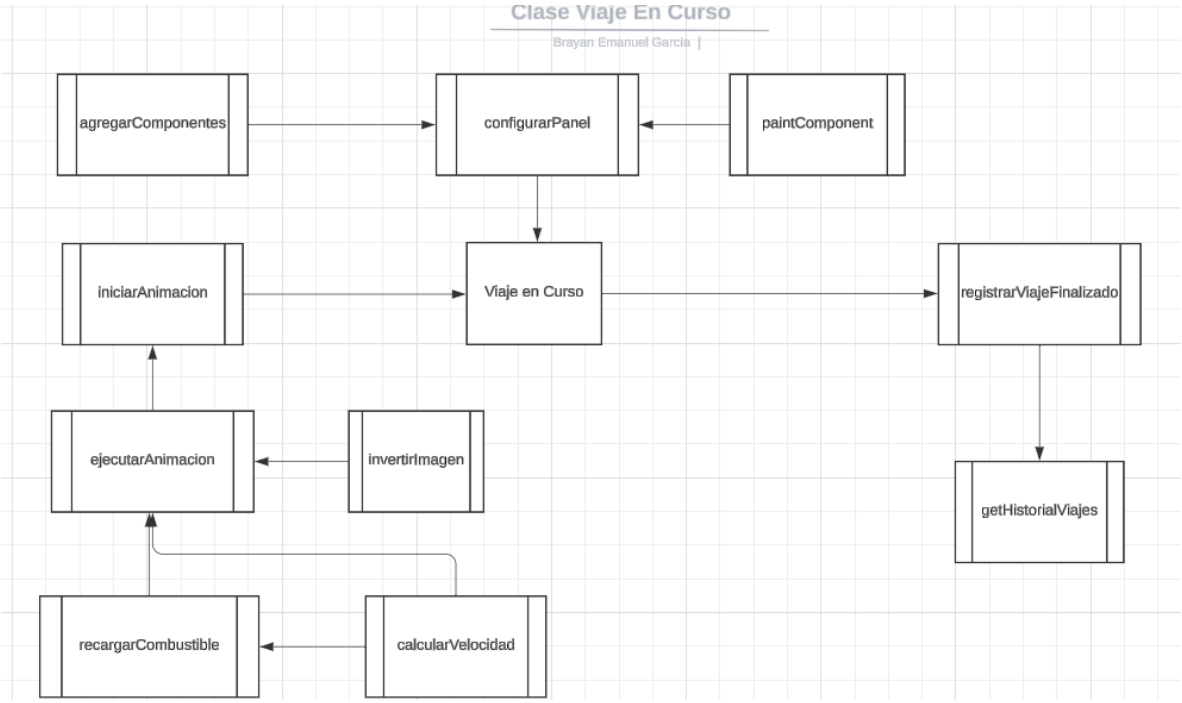


Diagrama de flujo

Brayan Emanuel García