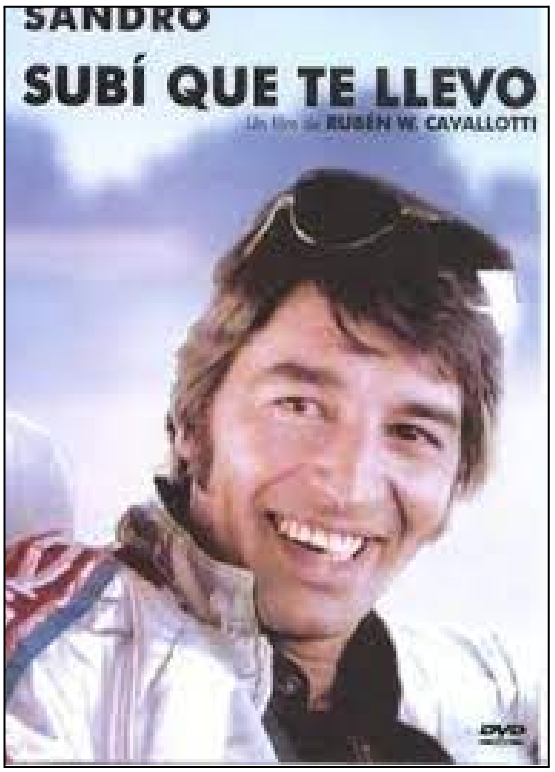
PROGRAMACIÓN III:

TRABAJO PRÁCTICO

ENTREGA FINAL

GRUPO 01



Mayor, Dario Agustín.

Nigro, Maite.

Patriarca, Martin.

Verón, Francisco.

INTRODUCCIÓN

Objetivos

1. Gestión de Vehículos, Clientes, Choferes y Viajes:

Crear un sistema que administre la información de la empresa, incluyendo la flota de vehículos, choferes y clientes registrados, permitiendo la adición de nuevos elementos a lo largo del tiempo.

2. Interfaz de Usuario en Computadora:

Desarrollar una interfaz de usuario accesible desde una computadora que sea intuitiva y funcional, facilitando la interacción con el sistema.

3. Creación y Evolución de Viajes:

Implementar la funcionalidad para que los clientes registrados puedan solicitar viajes a través de un formulario.
Gestionar la creación de viajes y permitir el seguimiento de su evolución, desde que se solicita hasta su finalización, de acuerdo con una cronología específica.

4. Patrones de Diseño y Arquitectura:

Aplicar el Patrón MVC (Modelo-Vista-Controlador) para estructurar el sistema, asegurando una separación clara entre la lógica de negocio, la interfaz de usuario y el control de flujo.
Utilizar los patrones Dao-Dto (Data Access Object - Data Transfer Object) para gestionar la persistencia de datos y la transferencia de información entre diferentes capas del sistema.
Implementar el patrón Observer/Observable para manejar eventos y notificaciones dentro del sistema.

5. Programación Concurrente:

Incluir programación concurrente y simulación mediante el uso de threads para manejar múltiples operaciones de forma simultánea.

6. Persistencia de Datos:

Asegurar que toda la información gestionada por el sistema (viajes, vehículos, choferes, clientes) sea persistente.

DESARROLLO

Diseño e implementación

Diagrama UML:

Se encuentra en: Grupo_01\MavenApp\TpFinalProgra3_2024\model\models

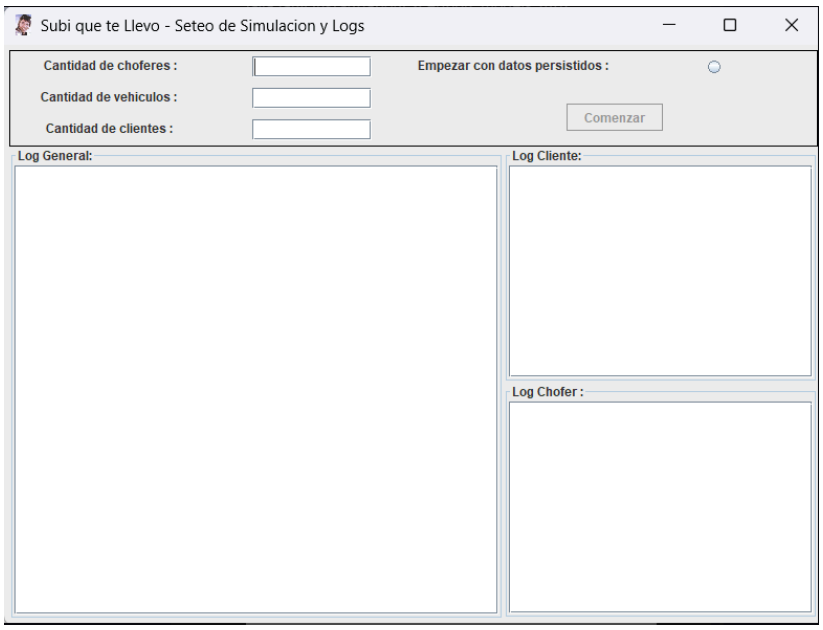
Patrón MVC

El patrón MVC ayuda a mantener todo ordenado dividiendo el sistema en tres partes. Con MVC, el sistema es más fácil de mantener y escalar, y cada parte se ocupa de lo que mejor sabe hacer, asegurando una experiencia de usuario eficiente y ordenada.

- Modelo: maneja los datos y la lógica de negocio como la información de los vehículos, choferes y clientes
- Vista: la interfaz de usuario donde los clientes pueden pedir viajes y ver su estado, y los administradores y choferes pueden gestionar recursos
- Controlador: actúa como intermediario, procesando las solicitudes de los usuarios, actualizando el Modelo y manteniendo la Vista al día

En este trabajo práctico se implementa el patrón MVC 2 veces, uno para la Ventana General y otro para la Venta de la AppCliente. Implementan una interfaz donde se definen en ella métodos para poder extraer datos ingresados por el usuario desde la ventana.
Cada ventana tiene su controlador homónimo. Implementa Actionlistener, para poder recibir los eventos que disparan las ventanas mediante los botones(por ejemplo, en la Ventana General, el botón **Comenzar**).

Ventana General:



El JFrame está dispuesto como *BorderLayout,* donde en su panel Norte se encuentran las opciones para iniciar la simulación, tanto con datos persistidos como por datos ingresados por el usuario.
El panel central cuenta con 3 *JTextArea* :

- LogGeneral: Se detallan todos los eventos que transcurren en la simulación.
- LogCliente: Se detalla los eventos de un cliente particular(en este caso, será el 1er cliente “Robot” creado).
- LogChofer:Se detalla los eventos de un chofer particular(en este caso, será el 1er chofer“Robot” creado).

Todos los datos ingresados son validados en la ventana:

- **No se pueden ingresar que no sean números en los campos Cantidad de choferes, Cantidad de vehículos y Cantidad de clientes.**

- **Si se escribe en alguno de los campos mencionados, el botón <u>Empezar con datos persistidos</u> se desactiva.**
- **Si el botón <u>Empezar con datos persistidos</u> se activa, se desactivan los campos de texto mencionados.**
- **Solo si una de estas opciones está activa, se puede presionar el botón <u>Comenzar</u>.**

Código destacable de VentanaGeneral:
Dentro de su constructor:

```
this.btnInicio.setActionCommand("Iniciar_Simulacion");
```

Se setea un Action command.Será un mensaje para el Controlador.
Fuera del constructor, se implementa el método setActionListener(está en la interfaz *IVistaGeneral*)

```
@Override  DarioM
public void setActionListener(ActionListener actionListener) {
    this.btnInicio.addActionListener(actionListener);
}
```

Interfaz *IVistaGeneral*:

```
public interface IVistaGeneral {  5 usages  1 implementation  DarioM +2 *
    public void setActionListener(ActionListener actionListener);

    public void appendLogCliente(String linea);  7 usages  1 implementa
    public void appendLogGeneral(String linea);  6 usages  1 implementa
    public void appendLogChofer(String linea);  4 usages  1 implementati

    public int getCantChoferes();  1 usage  1 implementation  DarioM
    public int getCantVehiculos();  1 usage  1 implementation  DarioM
    public int getCantClientes();  1 usage  1 implementation  DarioM
    public boolean isEnablePersistitdos();  1 usage  1 implementation
}
```

Métodos para ir actualizando los Logs de la ventana con los eventos que esten pasando en la simulación:

- appenLogCliente
- appendLogGeneral
- appendLogChofer

Métodos que extraen datos ingresados para luego crear la simulación:

- getCantChoferes
- getCantVehiculos
- getCantClientes

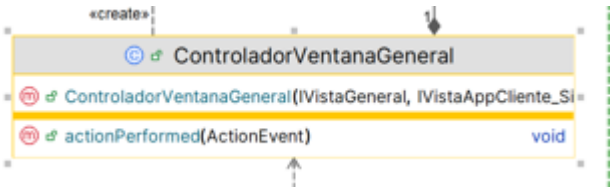
Método para saber si se va a utilizar persistencia o no:

- isEnablePersistidos

Método para definir quién es el ActionListener de la ventana.

Clase *ControladorVentanaGeneral*:

```
public class ControladorVentanaGeneral implements ActionListener
    private IVistaGeneral vista;  7 usages
    private IVistaAppCliente_SituacionViaje SV;  3 usages
```



Como atributos cuenta con dos interfaces: IVistaGeneral e IVistaAppCliente_SituacionViaje(Se analizará luego).

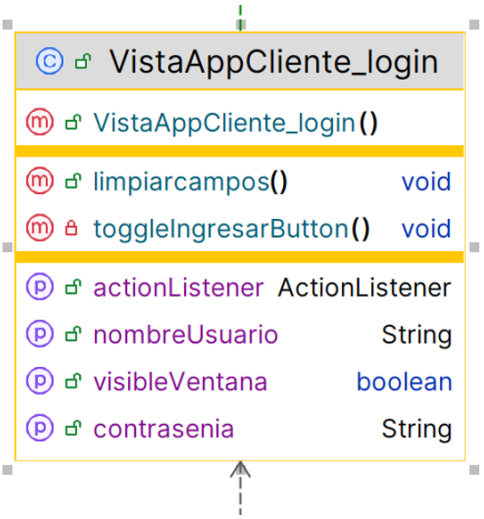
```
public ControladorVentanaGeneral(IVistaGeneral vista, IVistaAppCliente_SituacionViaje SV) {
    this.vista = vista;
    this.vista.setActionListener(this);
    this.SV = SV;
    this.SV.setActionListener(this);
}
```

Constructor del controlador de la Ventana. Lo destacable es que el Controlador se setea como ActionListener de las ventanas.

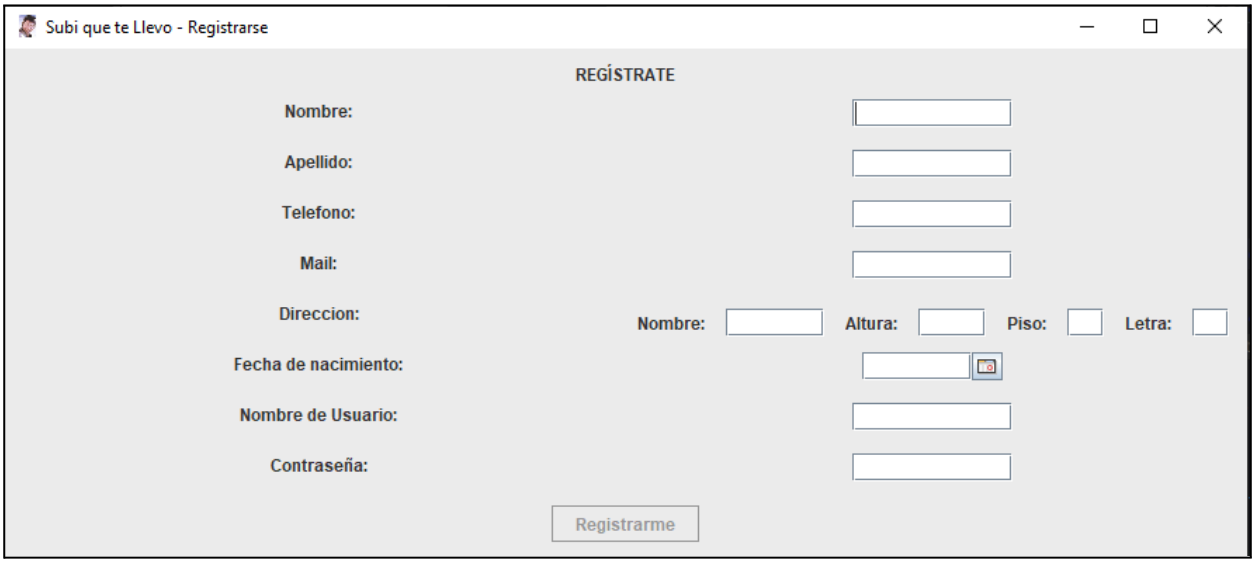
Luego, como método importante del Controlador es:
public void actionPerformed(ActionEvent e) {}. El controlador revisa qué comando se activó en la ventana, y luego lo resuelve. En este caso, revisa cuando se acciona el botón Comenzar. Luego revisa si se utiliza persistencia o no.

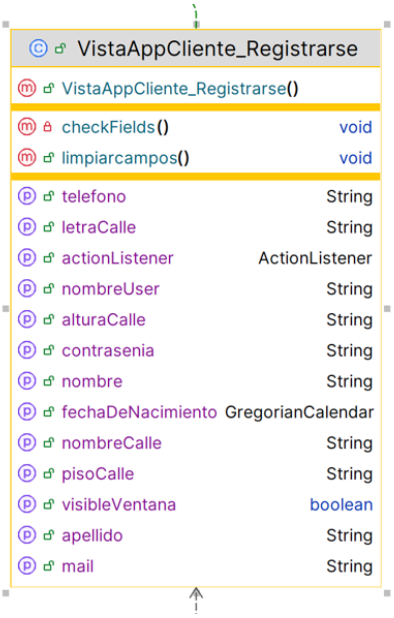
Ventana App Cliente:

En este caso, consta de 7 ventanas.
-VistaAppCliente_ login : Vista inicial de la App. Se ven 2 campos para ingresar Usuario y contraseña. Si los campos están vacíos, el botón Iniciar permanece desactivado. También hay un botón para registrarse. Si se ingresa un usuario o contraseña incorrectos, lanza una excepción avisando con un cartel.

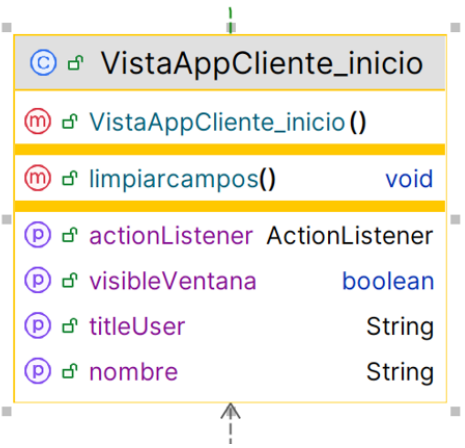


-*VistaAppCliente_Registrarse* : Vista para registrase de la app. Todos los campos deben estar completos, menos Piso y Letra. Si el nombre de usuario existe, lanza una excepción indicando que ese nombre ya fue usado.

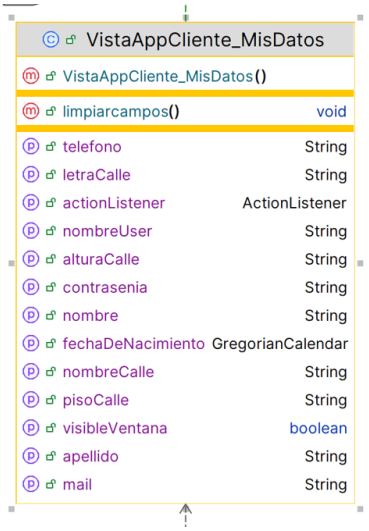




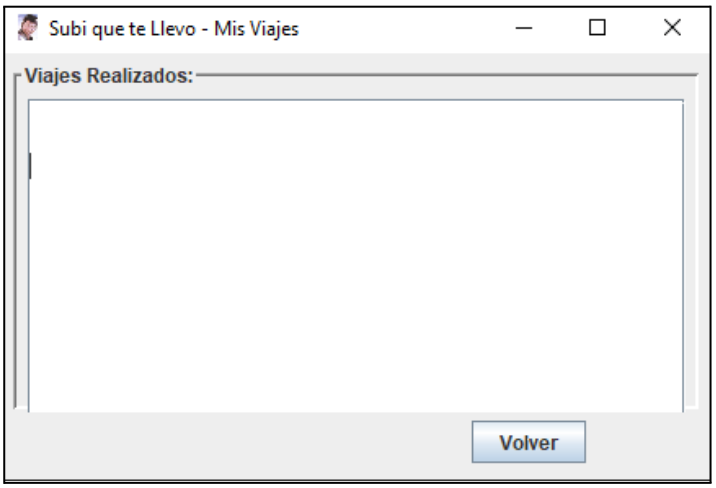
-*VistaAppCliente_Inicio* : Vista de inicio de la App. Muestra 4 botones: Hacer pedido, Mis datos, Mis viajes y un botón de Log Out.

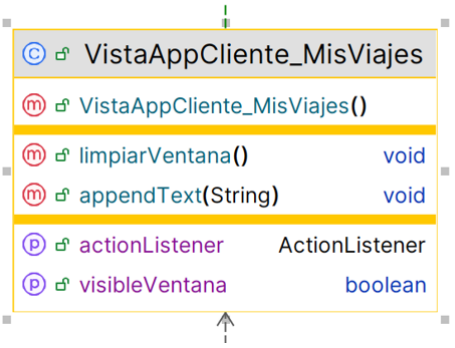


-VistaAppCliente_ MisDatos :Ventana de la App donde el usuario puede ver todos sus datos personales.

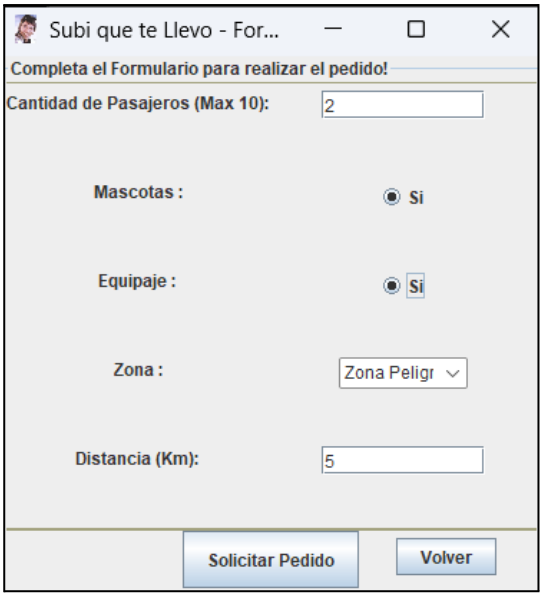


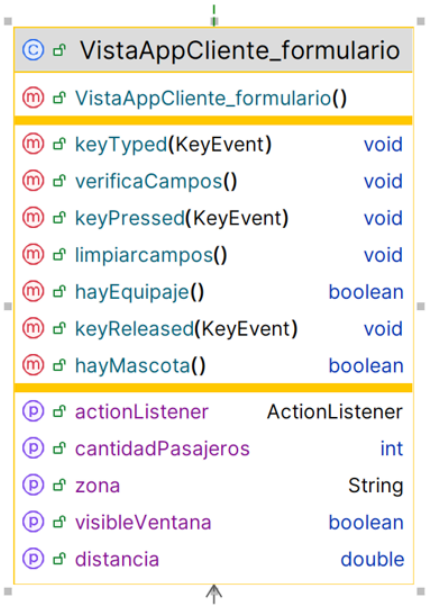
-VistaAppCliente_MisViajes : Ventana de la App donde el usuario puede ver los viajes realizados.



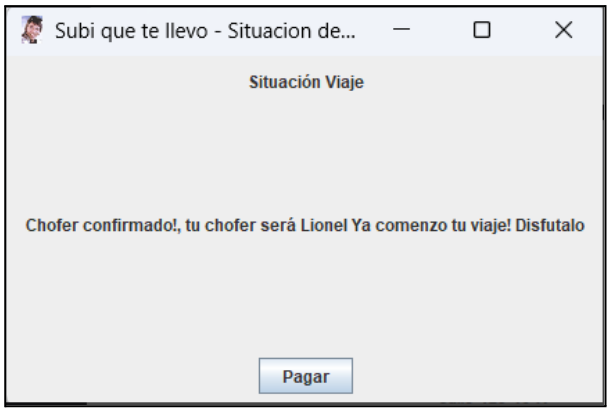


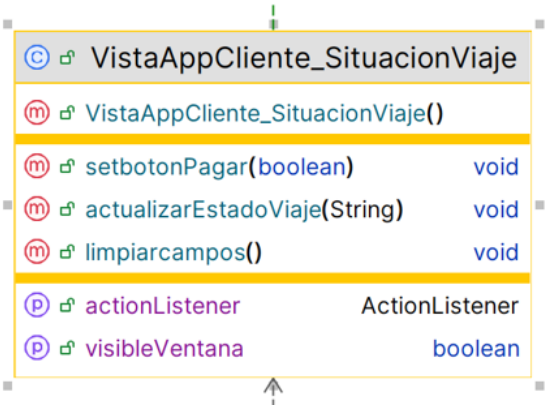
-VistaAppCliente_formulario : Ventana de la App donde el usuario llena el formulario del pedido. Se validan todos los datos como para que el usuario cree pedidos validados.





-*VistaAppCliente_SituacionViaje* : Vista de la App donde se ven en tiempo real el estado del viaje. Hay un botón de Pagar, solo se activa cuando el viaje está en estado Iniciado.





Todas las VistasApp validan los datos ingresados por el usuario. A su vez, tienen su interfaz homónima donde están los métodos para que el Controlador extraiga los datos necesarios y modifique datos en las ventanas.

Clase *Controlador* Vistas App:
Implementa ActionListener.

```
public class Controlador implements ActionListener {
    private IVistaAppCliente_formulario form;  12 usag
    private IVistaAppCliente_login login;  11 usages
    private IVistaAppCliente_inicio inicio;  14 usages
    private IVistaAppCliente_Registrarse r;  16 usages
    private IVistaAppCliente_SituacionViaje sv;  5 us
    private IVistaAppCliente_MisDatos md;  16 usages
    private IVistaAppCliente_MisViajes mv;  6 usages
```

Atributos del Controlador. Tiene las 7 interfaces de las vistas de la App.

```
public Controlador(IVistaAppCliente_formulario form,IV
    this.form = form;
    this.login = login;
    this.inicio = inicio;
    this.r = r;
    this.sv = sv;
    this.md =md;
    this.mv = mv;

    this.form.setActionListener(this);
    this.login.setActionListener(this);
    this.inicio.setActionListener(this);
    this.r.setActionListener(this);
    this.sv.setActionListener(this);
    this.md.setActionListener(this);
    this.mv.setActionListener(this);
```

Constructor de la clase. Lo importante es que el Controlador se setea como ActionListener de las vistas.
Luego, como método importante del Controlador es:
public void actionPerformed(ActionEvent e) {}. El controlador revisa qué comando se activó en las ventanas, y luego lo resuelve. Por ejemplo, en la ventana inicio, cuando se selecciona hacer pedido, el controlador se encarga de hacer visible una ventana y de no hacer visible la otra ventana.

```
else if (e.getActionCommand().equalsIgnoreCase( anotherString: "FORMULARIO_PEDIDOS"))
    this.form.setVisibleVentana(true);
    this.inicio.setVisibleVentana(false);
    this.inicio.limpiarcampos();
}
```

Partes destacables de las ventanas App:
-VentanaAppCliente_formulario:

```java
public void verificaCampos() {  3 usages  VeronFrancisco
    boolean camposValidos = true;

    // Verifica que la distancia no esté vacía y sea un número válido.
    try {
        camposValidos &= this.getDistancia() > 0;
    } catch (NumberFormatException e) {
        camposValidos = false;
    }

    // Verifica que la cantidad de personas no esté vacía, sea un número válido y esté en el rango permitido.
    try {
        int cantidadPasajeros = this.getCantidadPasajeros();
        camposValidos &= cantidadPasajeros > 0 && cantidadPasajeros <= 10;
        // Verifica que si hay mascota y el número de pasajeros es > 4, no se pueda hacer pedido
        if (this.hayMascota() && cantidadPasajeros > 4) {
            camposValidos = false;
        }
        // Desactiva el radio botón de mascota si hay más de 4 pasajeros
        this.rdbtnMascota.setEnabled(cantidadPasajeros <= 4);
        // Desactiva los radio botones de equipaje y mascota si hay solo 1 pasajero
        boolean habilitarRadioButtons = cantidadPasajeros != 1;
        this.rdbtnMascota.setEnabled(habilitarRadioButtons);
        this.rdbtnEquipaje.setEnabled(habilitarRadioButtons);
    } catch (NumberFormatException e) {
        camposValidos = false;
    }

    // Verifica que se haya seleccionado una zona.
    camposValidos &= !choice_zona.getSelectedItem().isEmpty();

    // Habilita o deshabilita el botón según la validez de los campos.
    btnPedir.setEnabled(camposValidos);
}
```

-VentanaAppCliente_inicio:

```java
@Override
public void setNombre(String nombre) {
    this.lbl_NombreUsuario.setText(nombre);
    this.lbl_NombreUsuario.setFont(new Font( name: "Tahoma", Font.BOLD,  size: 16));
}
```

Hace que la app sea más personal, muestra en la ventana de inicio el nombre de usuario. Es un detalle estético más que nada.

En varias ventanas se implementa este método:

```java
@Override  VeronFrancisco
public void limpiarcampos() {
    this.dateChooser.cleanup();
    this.tf_alturaCalle.setText("");
    this.tf_letraCalle.setText("");
    this.tf_PisoCalle.setText("");
    this.tf_nombreCalle.setText("");
    this.tl_apellidonuevo.setText("");
    this.tl_telefononuevo.setText("");
    this.tl_nombrenuevo.setText("");
    this.tl_mailnuevo.setText("");
    this.tl_contrasenianuevo.setText("");
    this.tl_Usuarionuevo.setText("");
}
```

Es una forma para poder limpiar los campos de la ventana cuando nos vamos, por si queremos luego volver a ella.

Patrones Utilizados

Patron Singleton

El patrón Singleton es de diseño creacional y tiene como objetivo asegurarse que una clase tiene una única instancia de acceso global.
En nuestro programa, es utilizado con la clase Sistema con el objetivo de tener una consistencia en la gestión de la lógica y coordinación de la aplicación y tener la posibilidad del acceso global (ya que es la encargada de iniciar los viajes y llevar un registro del mismo en todo momento)

```java
public static Sistema getInstancia()
    if (_instancia == null)
        _instancia = new Sistema();
    return _instancia;
}
```

Patron Factory

Es un patrón de diseño creacional que proporciona una interfaz para crear objetos en una clase base, pero permite a las subclases alterar el tipo de objetos que se crean. Lo utilizamos en dos casos.

El primero de ellos (IVehiculos), es para los vehículos, que define métodos específicos para crear diferentes tipos de vehículos que implementan esta interfaz.

```
public interface  IVehiculo extends Cloneable, Serializable  {  4 implementations  DarioMayor +2
  Integer getPrioridad(Pedido pedido);  1 usage  4 implementations  DarioMayor
  boolean validarVehiculo(Pedido pedido);  4 usages  1 implementation  DarioMayor
  Vehiculo clone()throws CloneNotSupportedException;  1 implementation  DarioMayor
  public void setOcupado(boolean x);  1 implementation  DarioMayor
  public boolean isOcupado();  1 implementation  DarioMayor
}
```

Por lo que luego, tendremos instancias de Vehículos(tutu, motos y combis) utilizando los métodos implementados en la interfaz IVehiculo

```
    public static IVehiculo getVehiculo(String vehiculo,String patente) {  3 usages  VeronFrancisco
        IVehiculo encapsulado = null;
        if (vehiculo.equalsIgnoreCase( anotherString: "auto"))
            encapsulado = new Tutu(patente);
        else if (vehiculo.equalsIgnoreCase( anotherString: "combi")) {
            encapsulado = new Combi(patente);
        }
        else if (vehiculo.equalsIgnoreCase( anotherString: "moto")) {
            encapsulado = new Moto(patente);
        }
        return encapsulado;
    }
}
```

Clase de tipo IVehiculo utilizada para permitir crear vehículos (auto,combi o moto) sin necesidad de conocer los detalles específicos de su implementación

El segundo caso en que lo usamos es con una interfaz de viajes “IViaje”

```
public interface IViaje extends Cloneable, Serializable {
 double getCosto_viaje();
 void setCosto_viaje(double valor);
 Pedido getPedido();
 void setPedido(Pedido pedido);
 void setEstado_de_viaje(String estado_de_viaje);
 String getEstado_de_viaje();
 Empleado getChofer();
 void setChofer(Empleado chofer);
 IVehiculo getVehiculo();
 void setVehiculo(IVehiculo vehiculo);
 void setCosto_base(double costo_base);
  double getCosto_base();
  void calcularCostoViaje();

  void pagarse();
  void  finalizarse();
  IViaje clone() throws CloneNotSupportedException;
  int compareTo(Object o);

}
```

En este caso, utilizamos el patrón para poder crear varios viajes que cumplan con los métodos planteados en la interfaz pero que cada uno de ellos los puedan implementar de manera diferente y que cada viaje pueda ser personalizado según las características del mismo solicitado por el cliente.

```
public static IViaje getViaje(Pedido pedido) {  1 usage   Maite2003
    IViaje respuesta = new Viaje(pedido);

    // Genero viaje segun su zona
    if (pedido.getZona().equalsIgnoreCase( anotherString: "SIN ASFALTAR"))
        respuesta = new DecoratorZonaSinAsfaltar(respuesta);
    if (pedido.getZona().equalsIgnoreCase( anotherString: "ESTANDAR"))
        respuesta = new DecoratorZonaEstandar(respuesta);
    if (pedido.getZona().equalsIgnoreCase( anotherString: "PELIGROSA"))
        respuesta = new DecoratorZonaPeligrosa(respuesta);

    if (pedido.isMascota()) respuesta = new DecoratorMascota(respuesta);

    if (pedido.isEquipaje()) respuesta = new DecoratorEquipaje(respuesta);

    return respuesta;
}
}
```

Método de tipo IViaje que averigua las condiciones del viaje solicitado y así genera el viaje según su zona y especificaciones de mascota y equipaje

Patron Template

Este patrón es de tipo comportamiento, se trata en permitir que ciertos pasos de un algoritmo definido en un método de una clase sea redefinido en sus clases derivadas sin necesidad de sobrecargar la operación entera. Teniendo una superclase con subclases que comparten métodos pero son utilizados de diferente manera, el patrón permite heredar el método de su superclase y sobreescribirlo según la necesidad de cada clase hija. Lo podemos ver aplicado al asignarle una prioridad a los vehículos, que varía según si cumple con los requisitos del pedido (Cantidad de pasajeros, uso de baúl, pet friendly)].

Superclase: VEHICULO

```
public Integer getPrioridad(Pedido pedido) {
    int puntaje;
    if (!(validarVehiculo(pedido)))
        return null;
    if (pedido.isEquipaje())
        puntaje = 40 * pedido.getCant_pasajeros();
    else
        puntaje = 30;
    return puntaje;
}
```

Mientras que en las subclases podemos ver este mismo método “getPrioridad” sobreescrito:

Subclase tutu:

```java
@Override 1 usage  DarioM +2
public Integer getPrioridad(Pedido pedido) {
    int puntaje;
    if (!(validarVehiculo(pedido)))
        return null;
    if (pedido.isEquipaje())
        puntaje = 40 * pedido.getCant_pasajeros();
    else
        puntaje = 30;
    return puntaje;
}
```

Subclase moto:

```java
@Override  Maite2003 +1
public Integer getPrioridad(Pedido pedido) {
    int puntaje = 0;
    if (!(validarVehiculo(pedido))) return null;
    if (pedido.getCant_pasajeros() == 1 && !pedido.isEquipaje() && !pedido.isMascota()) puntaje = 1000;
    return puntaje;
}
```

Subclase combi:

```java
@Override  1 usage  DarioMayor +1
public Integer getPrioridad(Pedido pedido) {
    int puntaje;
    if (!(validarVehiculo(pedido))) return null;
    puntaje = pedido.getCant_pasajeros()*10;
    if (pedido.isEquipaje()) puntaje += 100;
    return puntaje;
}
```

Patron Observer/Observable:

Observer: Define una dependencia entre objetos de modo que, cuando un objeto cambia su estado, todos sus dependientes son notificados y actualizados automáticamente. Se puede utilizar este patrón cuando un objeto es dependiente de otro, cuando tras un cambio de estado se requiere que se actualicen otros y cuando un objeto debería ser capaz de notificar a otros sobre sus cambios.
Observable: Cada objeto Observable mantiene un indicador de “modificado” que es utilizado por los métodos de la subclase para indicar que ha sucedido algo de interés. Cuando ocurre un cambio, el objeto Observable debe invocar al método setChanged y después notificarlo a los observadores.

En nuestro programa se ve aplicado en la clase *BolsadeViajes* (clase observable que puede mantener una lista de observadores y notificarles sobre cambios), donde, según el método llamado, el viaje cambia su estado y es notificado a SistemaRunnable, su Observador(Esto permite que el Sistema responda a notificaciones de BolsaDeViajes).

```
@Override  FranciscoVeronING +2
public void update(Observable obs, Object arg) {
    if (obs != this.bolsaViajes) {
        throw new IllegalArgumentException();
    }
    synchronized (this) {
        EventoSistema evento = (EventoSistema) arg;
        if (evento.getMensaje().equalsIgnoreCase(EventoSistema.NUEVOVIAJE) || evento.getMensaje().equalsIgnoreCase(EventoSistema.STOP)) {
            this.eventos.add(evento);
        }
    }
}
```

En el método update de SistemaRunneable si el evento que le observa es del tipo NUEVOVIAJE o del tipo STOP, inserta el evento en su cola de eventos.

```
public void run() {  Maite2003 +3
    while (bolsaViajes.isSimulacionActiva()) { // Hasta q se pare la simulacion
        if (!eventos.isEmpty()) { // Cuando recibe un evento de nuevo viaje
            IViaje viaje = eventos.poll().getViaje();
            if (viaje != null && !vehiculosDisponibles.isEmpty()) {
                IVehiculo vehiculoAsignado = getVehiculoValido(viaje);
                if (vehiculoAsignado != null) {
                    bolsaViajes.asignarVehiculo(viaje, vehiculoAsignado);
                    viaje.setVehiculo(vehiculoAsignado);
                }
                else {
                    viaje.setEstado_de_viaje("RECHAZADO");
                    bolsaViajes.rechazarViaje(viaje);
                }
            }
        }
    }
    Sistema.getInstancia().cargaSistema();
}
```

En el método run(), SistemaRunenable trata los eventos de la cola de eventos.

También hay un Observador encargado de modificar los Logs de las ventanas, es decir, le informa a las ventanas los eventos recientes de la simulación. Observa a los Threads Clientes, Choferes, y ClienteApp. Modifica el LogGeneral, LogCliente, LogChofer y la ventana VistaAppCliente_SituacionViaje.

```
public class ObservadorVentanaGral extends ObservadorAbstracto {
    private IVistaGeneral vistaG;  28 usages
    private IVistaAppCliente_SituacionViaje vistaSV;  12 usages
    private Cliente clienteRobot;  2 usages
    private Cliente clienteApp;  2 usages
    private Empleado chofer;  2 usages
```

Tiene 3 flags de tipo boolean para saber que tipo de hilo está observando. Otros atributos son las 2 ventanas con las que se comunica, Vista general y VistaAppCliente_SituacionViaje.
El método update(Observable obs, Object arg) se encarga de capturar los eventos de la simulación y revisa de qué tipo es el hilo que lo mandó, así actuando en consecuencia.

Fragmento del código de update:

```
Cliente c = null;
if (evento.getViaje() != null) c = evento.getViaje().getPedido().getCliente();
else if (evento.getPedido() != null) c = evento.getPedido().getCliente();

boolean esClienteRobot = c == clienteRobot;
boolean esClienteApp = c == clienteApp;
boolean esChofer = false;
if (evento.getViaje() != null && evento.getViaje().getChofer() != null) {
    esChofer = chofer == evento.getViaje().getChofer();
}

if (mensaje.equalsIgnoreCase(EventoSistema.NUEVOPEDIDO)) {
    assert c != null;
    this.vistaG.appendLogGeneral( linea: "//// Usuario "+ c.getNombre_usuario() + " creó un pedido\n");
    if (esClienteRobot) this.vistaG.appendLogCliente( linea: "Creó un pedido\n");
    if (esClienteApp) this.vistaSV.actualizarEstadoViaje( nuevoEstado: "Creaste un pedido! Esperando confirmacion");
```

En las últimas 2 líneas se verifica quién envió el evento, si fue el cliente robot o el clienteApp, luego se actúa en consecuencia. Siempre el evento se muestra por el Log general.

Patrón DTO

Es un patrón de diseño que se utiliza para transferir datos entre diferentes componentes de una aplicación. Tiene como finalidad la creación de un objeto plano con una serie de atributos que puedan ser enviados o recuperados del servidor en una sola invocación. El patrón tiene dos reglas que deben cumplirse:

> Solo lectura: ya que el objetivo de un DTO es ser utilizado como un objeto de transferencia entre el cliente y el servidor.
>
> Serializable: No solo la clase DTO debe ser serializable, sino todas aquellas que también tiene todos los atributos que contengan el DTO.

Utilizamos este patrón en la clase “SistemaDTO” que implementa la interfaz Serializable. Esta clase se está utilizando para encapsular y transportar datos sobre choferes, vehículos, clientes y la bolsa de viajes entre diferentes partes del sistema.

```
public void cargaSistema(){  1 usage   DarioMayor
    SistemaDTO sistemaDTO= new SistemaDTO();
    sistemaDTO = SistemaXML.cargaSistema();
    Sistema.getInstancia().setChoferes(sistemaDTO.getChoferes());
    Sistema.getInstancia().setVehiculos(sistemaDTO.getVehiculos());
    Sistema.getInstancia().setClientes(sistemaDTO.getClientes());
    Sistema.getInstancia().setClientesApp(sistemaDTO.getClientesApp());
    Sistema.getInstancia().setClientesActivos(sistemaDTO.getClientes().size());
    Sistema.getInstancia().setChoferesActivos(sistemaDTO.getChoferes().size());
}
```

```java
public static SistemaDTO sistemaDTOfromSistema(){  1 usage  FranciscoVeronINC
    SistemaDTO sistemaDTO = new SistemaDTO();
    sistemaDTO.setChoferes(Sistema.getInstancia().getChoferes());
    sistemaDTO.setVehiculos(Sistema.getInstancia().getVehiculos());
    sistemaDTO.setClientes(Sistema.getInstancia().getClientes());
    sistemaDTO.setClientesApp(Sistema.getInstancia().getClientesApp());
    return sistemaDTO;
}
```

El método sistemaDTOfromSistema utiliza el patrón DTO al cargar un objeto con los datos del sistema encapsulando los datos y pudiendo transferirlos entre las diferentes del programa

```java
public class SistemaXML {  4 usages  DarioMayor +1
    private static final String File_Name = "SISTEMA.xml";  2 usages
    public static void grabaSistema() {  1 usage  DarioMayor +1
        SistemaDTO sistemaDTO = UtilSistema.sistemaDTOfromSistema();
        XMLEncoder encoder = null;
        try{
            encoder = new XMLEncoder(new BufferedOutputStream(new FileOutputStream(File_Name)));
        }
        catch(FileNotFoundException fileNotFound){
            System.out.println("ERROR AL CREAR EL ARCHIVO");
        }
        assert encoder != null;
        encoder.writeObject(sistemaDTO);
        encoder.close();
    }

    public static SistemaDTO cargaSistema() {  1 usage  DarioMayor +1
        XMLDecoder decoder = null;
        try{
            decoder = new XMLDecoder(new BufferedInputStream(new FileInputStream(File_Name)));
        }
        catch(FileNotFoundException e){
        }

        assert decoder != null;
        SistemaDTO sistemaDTO = (SistemaDTO) decoder.readObject();
        return sistemaDTO;
    }
}
```

El método cargaSistema utiliza el patrón DTO al cargar un objeto desde un archivo XML encapsulando los datos y pudiendo transferirlos entre diferentes partes del sistema

Diseños de Clases

- **Excepciones:**

ChoferNoDisponibleException*:* Excepcion que lanzamos cuando todos los choferes están ocupados y no se le puede asignar un pedido a un chofer porque no hay ninguno disponible
PedidoImposibleException*:* Excepcion padre que lanzamos cuando no es posible tomar un pedido por diferentes motivos como falta de vehículo,chofer o pedido incoherente
PedidoIncoherenteException*:* Excepción que lanzamos cuando el pedido solicitado por el cliente es incoherente en relación a los servicios brindados por la empresa
UsuarioRepetidoException*:* Excepción que lanzamos cuando un usuario nuevo se quiere registrar con un nombre de usuario que ya está siendo utilizado por otro usuario
UsuarioIncorrectoException: Excepcion que lanzamos cuando un usuario nuevo no cumple con los requisitos de validación a la hora de querer crear un usuario nuevo.
VehiculoNoDIsponibleException*:* Excepcion que lanzamos cuando todos los vehiculos están siendo usados y no hay ninguno disponible para seguir tomando pedidos

- **Interfaces**:

IVehiculo*:* Interfaz que define los métodos que la clase Vehículo debe implementar

Define el contrato para los vehículos que pueden ser clonados y serializados.

IViaje*:* Interfaz que define los métodos que la clase viaje debe implementar

La interfaz IVehiculo define el contrato para los vehículos que pueden ser clonados y serializados.

IVistaAppCliente_formulario: Interfaz que define los métodos que la clase Vehículo debe implementar

IVistaAppCliente_inicio: Interfaz que define los métodos que la clase VistaAppCliente_inicio debe implementar
IVistaAppCliente_login: Interfaz que define los métodos que la clase VistaAppCliente_login debe implementar
IVistaAppCliente_MisDatos: Interfaz que define los métodos que la clase VistaAppCliente_MisDatos debe implementar
IVistaAppCliente_MisViajes: Interfaz que define los métodos que la clase VistaAppCliente_MisViajes debe implementar
IVistaAppCliente_Registrarse: Interfaz que define los métodos que VistaAppCliente_Registrarse debe implementar
IVistaAppCliente_SituacionViajes: Interfaz que define los métodos que la clase VistaAppCliente_SituacionViajes debe implementar
IVistaGeneral: Interfaz que define los métodos que la clase VistaGeneral debe implementar

- **Clases**:

***Sistema**:* Representa a la empresa, se utiliza una única instancia de esta clase aplicando el patrón SIngleton donde se encarga de evaluar el pedido y definir si lo acepta o lo rechaza

Administrador: Clase a utilizar para cuando el usuario se registra como administrador.

***Empleado**:* Es la clase que representa a todo empleado de la empresa.

***Chofer**:* Clase abstracta que representa a los choferes de la empresa y se extiende de la clase empleado

***ChoferContratado**:* Clase extendida de la clase Chofer y representa a los choferes de tipo Contratado

***ChoferPermanente**:* Clase extendida de la clase Chofer y representa a los choferes de tipo Permanente

***ChoferTemporario**:*Clase extendida de la clase Chofer y representa a los choferes de tipo Temporario

***Usuario**:* Clase abstracta que permite usuarios nuevos en el sistema

***Cliente**:* Clase que representa el cliente y permite crear un nuevo cliente en el sistema con toda su información.

***Dirección**:* Clase que representa la dirección donde vive el cliente.

***VehiculoFactory**:*Clase utilizada para verificar si el vehículo a utilizar será un auto,combi o moto

***Vehiculo**:* La clase implementa la interfaz Vehículo y es la clase padre de todos los vehículos que tiene la empresa

***Tutu**:* Clase que se extiende de Vehículo y es utilizada para calcular la prioridad que tendrá el **auto** a la hora de escoger un vehículo

***Moto**:* Clase que se extiende de Vehículo y es utilizada para calcular la prioridad que tendrá la moto a la hora de escoger un vehículo

***Combi**:* Clase que se extiende de vehículo y es utilizada para calcular la prioridad que tendrá la combi a la hora de escoger un vehículo.

***DecoratorEquipaje**:* Clase que se usa para agregar funcionalidades adicionales al equipaje de manera dinámica.

***DecoratorMascota**:* Clase que se usa para agregar funcionalidades adicionales a la mascota de manera dinámica.

***DecoratorZonas**:* Clase padre que se usa para agregar funcionalidades adicionales a las zonas de manera dinámica.

***DecoratorZonaEstandar**:* Clase que se extiende de DecoratorZonas y se usa para agregar funcionalidades adicionales a la zona estándar de manera dinámica.

***DecoratorZonaPeligrosa**:* Clase que se extiende de DecoratorZonas y se usa para agregar funcionalidades adicionales a la zona peligrosa de manera dinámica.

***DecoratorZonaSinAsfaltar**:* Clase que se extiende de DecoratorZonas y se usa para agregar funcionalidades adicionales a la zona sin asfaltar de manera dinámica.

***Pedido**:* Es la clase que permite al usuario elegir sus restricciones a la hora de buscar un vehículo apropiado para su viaje

***Viaje**:* La clase implementa IViaje y es utilizada para qué armar el viaje apropiado para el usuario, seleccionado un chofer,vehiculo, marcando el viaje como solicitado

***ViajeFactory**:* Clase utilizada para aplicar el patrón Factory a la clase Viaje

ClienteRunnable: Clase que se extiende de Clientes e implementa Runnable. Da inicio al hilo del cliente donde se crea el pedido y solicita que sea aceptado

EmpleadoRunnable: Clase que se extiende de Empleado e implementa Runnable. Da inicio al hilo del empleado utilizando el método de wait/notify para cuando el mismo se ocupa y se desocupa una vez que el estado del viaje sea “pagado”.

SistemaRunnable: Clase que se extiende de Threads e implementa Observer. Da inicio al hilo del Sistema verificando que hayan viajes y/o vehículos disponibles. Luego asigna un vehículo y agrega el viaje a la bolsa de viajes.

SistemaDTO: Encapsula y transfiere datos del sistema relacionado con choferes, vehículos, clientes y viajes.

SistemaXML: Guarda y carga el estado del sistema en un archivo XML.

EventoSistema: Clase utilizada para indicar el estado en el que se encuentra el viaje.

Utiles: Clase útil para crear datos de prueba o ejemplos de manera aleatoria.

UtilSistema: Clase de utilidad que proporciona métodos estáticos para interactuar con el sistema.

- **Vista:**

Controlador: Es una clase que implementa la interfaz ActionListener y maneja las interacciones del usuario con las vistas de la aplicación.

ControladorVentanaGeneral: Controlador en un patrón de diseño MVC (Modelo-Vista-Controlador) que se encarga de gestionar las acciones provenientes de una vista general

ObservadorAbstracto: Es una clase abstracta que implementa la interfaz Observer y actúa como un observador genérico en el patrón de diseño Observador

Observador: La clase actúa como un observador específico para los eventos relacionados

VistaAppCliente_formulario : La clase representa la interfaz gráfica para que los clientes completen un formulario y realicen un pedido en la aplicación

VistaAppCliente_inicio: Representa la interfaz gráfica de la ventana de inicio de sesión para los clientes en la aplicación.

VistaAppCliente_login: Representa la interfaz gráfica de la ventana de inicio de sesión de los clientes en la aplicación

VistaAppCliente_MisDatos: Representa la interfaz gráfica de la ventana que muestra los datos personales del cliente en la aplicación.

VistaAppCliente_MisViajes: Representa la interfaz gráfica de la ventana que muestra los viajes realizados por el cliente en la aplicación.

VistaAppCliente_Registrarse: Representa la interfaz gráfica de la ventana de registro para nuevos clientes en la aplicación

VistaAppCliente_SituacionViaje: Representa la interfaz gráfica para mostrar la situación de un viaje en la aplicación

VistaGeneral: Representa la interfaz gráfica para configurar y controlar la simulación de un sistema de transporte

- **JavaDoc:**

Se encuentra en la carpeta del proyecto(**GitHub**).

Dificultades y soluciones

Nos enfrentamos a desafíos tanto técnicos como relacionados con la dinámica grupal. Tuvimos dificultades al principio con la implementación del patrón FACADE, debido a que fue una corrección de la anterior entrega, la programación concurrente, la persistencia de datos y las ventanas.

Las ventanas fueron un desafío, no por la parte estética, sino por la comunicación en “tiempo real” con el modelo. Para la ventana “VistaAppCliente_SituaciónViaje” se pensó que el usuario viera en tiempo real cómo iba su estado de viaje, desde Pedido realizado con éxito hasta Viaje pagado. No nos dábamos cuenta de cómo notificar al Controlador el cambio de estado del viaje.

Se resolvió haciendo que el Observador sea el encargado de modificar en tiempo real la ventana, teniendo como atributo a esta vista. Luego, se modificó el observador(antes eran 4 Observadores, se pasó a tener solo 1) y, dependiendo el Thread que sea es el flag que se habilita. En caso de que sea el clienteApp va a llamar al método **this.vistaSV.actualizarEstadoViaje(“** mensaje dependiendo el Evento que le mande**”),** así actualizando el estado del viaje en la ventana. Ideas como tener un observador aparte que le notifique al controlador fueron descartadas.

Extracto de código del Observador, dentro del método update(Observable obs, Object arg):

```
if (mensaje.equalsIgnoreCase(EventoSistema.NUEVOPEDIDO)) {
    this.vistaG.appendLogGeneral( linea: "&&&& Usuario "+ c.getNombre_usuario() + " creó un pedido\n");
    if (esClienteApp) this.vistaSV.actualizarEstadoViaje( nuevoEstado: "Creaste el pedido");
    if (esClienteRobot) this.vistaG.appendLogCliente( linea: "&&&& El cliente creo un pedido\n");
```

Con respecto a los problemas con la programación recurrente hubo problemas con respecto de cómo era el recorrido de los Threads. Primero se crearon todas los métodos necesarios y no funcionaba la concurrencia. Nos dimos cuenta que nos olvidamos de poner synchronized en los métodos que necesitaban ser sincronizados. Luego los hilos nunca se iniciaban, nos dimos cuenta que nos olvidamos de instanciar los Threads cuando se creaban clientes/choferes. Lo mismo para el Thread Sistema. Estos errores fueron más de distraídos. Algo que no fue un error, sinó una modificación es que habíamos puesto un botón de Finalizar Simulación en la Ventana General, lo cual finaliza la simulación cuando se accionaba. Se descartó porque fueron modificados los requerimientos del trabajo práctico, haciendo que los clientes tengan una cantidad de pedidos X(lo resolvimos creando una cantidad random entre 0 y 4). La simulación termina cuando se apriete el botón Finalizar Simulación el la *VistaAppCliente_inicio*.

Fragmento del método actionPerformed() de la clase Controlador:

```
else if(e.getActionCommand().equalsIgnoreCase( anotherString: "FINALIZAR")) {
    Sistema.getInstancia().guardaSistema();
    Sistema.getInstancia().detenerSimulacion();
}
```

Cuando se acciona el botón Finalizar Simulación, se llama al método guardaSistema(),en donde se persisten los datos de la simulación. Luego se llama al método detenerSimulación(), lo que hace es poner en ***false*** el flag de simulacionActiva.

Método guardaSistema(), se encarga de realizar la persistencia de los datos.

```
public void guardaSistema(){sistemaOutput.grabaSistema();}
```

Método detenerSimulacion(), se encarga de notificar al observador que el sistema se detuvo.

```
public void detenerSimulacion() { this.viajes.detenerSimulacion(); }
```

```
public synchronized void detenerSimulacion() {  1 usage   DarioMayor
    this.simulacionActiva = false;
    notifyAll();
    setChanged();
    notifyObservers(new EventoSistema(EventoSistema.STOP));
}
```