PROGRAMACIÓN III

TP FINAL

“SUBÍ QUE TE LLEVO”

INTEGRANTES:

- Ahumada, Ivan.
- Magliotti, Gian Franco.
- San Pedro, Gianfranco.

Fecha:9/06/24

ÍNDICE

INTRODUCCIÓN

La finalidad del presente trabajo es brindar una continuación del informe anterior perteneciente a la primera etapa del trabajo final de la materia Programación III, Facultad de Ingeniería , UNMDP.

El objetivo principal de dicho trabajo fue modelar una app de servicios de remises. Esta segunda parte contó con el agregado de la concurrencia,de forma tal de simular comportamientos más realistas en cuanto al acceso de los recursos, y con el desarrollo de una interfaz de usuario que permita al humano interactuar como si fuese un cliente real de la empresa.

Durante el desarrollo del mismo, se abordan explicaciones detalladas acerca de las implementaciones, patrones de diseño, clases y metodologías empleadas para poder resolver el trabajo.

Cada una de las explicaciones cuenta con gráficos que sirven de guía y justificaciones en su mayoría fundamentadas en la teoría vista a lo largo del cuatrimestre.

METODOLOGÍA

Para desarrollar el trabajo, se empleó una metodología de trabajo grupal diferente a la estándar. En lugar de separar los distintos aspectos a implementar de la aplicación individualmente, decidimos ir tomando cada decisión en conjunto, de forma tal de evitar los conflictos usuales que ocurren al querer agrupar soluciones de enfoques completamente diferentes para obtener un todo final.

De esta manera, aseguramos un avance firme, tal vez más lento, pero sin lugar a dudas más efectivo. Abordar cada aspecto del diseño de manera grupal, permitió que contáramos con opiniones variadas en torno a la solución de un aspecto, en lugar de cerrar las opciones a la de la persona que le hubiese tocado desarrollar esa parte.Además, permite también identificar errores tempranamente antes que escalaran o se vieron descubiertos con el código ya avanzado.

Por último, destacamos el importante uso de una implementación en pseudocódigo previa a la implementación real. Esto nos ayudó a coordinar temas complejos como la concurrencia y a la vez se evitó el típico error de comenzar a codificar la primera solución que surge. Tal manera de actuar, derivó en que a medida que avanzamos en la solución, no tuviéramos que volver atrás para arreglar un error de arrastre.

Motivación

Antes de comenzar con el desarrollo propiamente dicho, se procede a resaltar algunas de las motivaciones que el grupo encontró a la hora de desarrollar el trabajo:

- Obtener la posibilidad de aplicar los conceptos teóricos estudiados sobre el paradigma de la programación orientada a objetos a un caso realista
- Aprender a tomar decisiones más allá del punto de vista algorítmico que veníamos trabajando, ampliándolo a un nivel de diseño que implica entender los requerimientos del sistema y saber delegar responsabilidades entre las diferentes clases
- Aprender patrones de diseño que puedan sernos útiles en proyectos personales o profesionales a futuro
- Experimentar el desarrollo de una “aplicación” desde la definición de las clases y funcionalidades hasta la implementación de una interfaz gráfica
- Ganar experiencia en el uso del lenguaje de programación JAVA, conocer las ventajas que nos ofrece respecto a otros como C para llevar adelante un trabajo como el presente

Las motivaciones anteriormente listadas, solo resaltan algunos de los puntos que consideramos más importantes. Sin embargo, también, nos encontramos con el desafío de

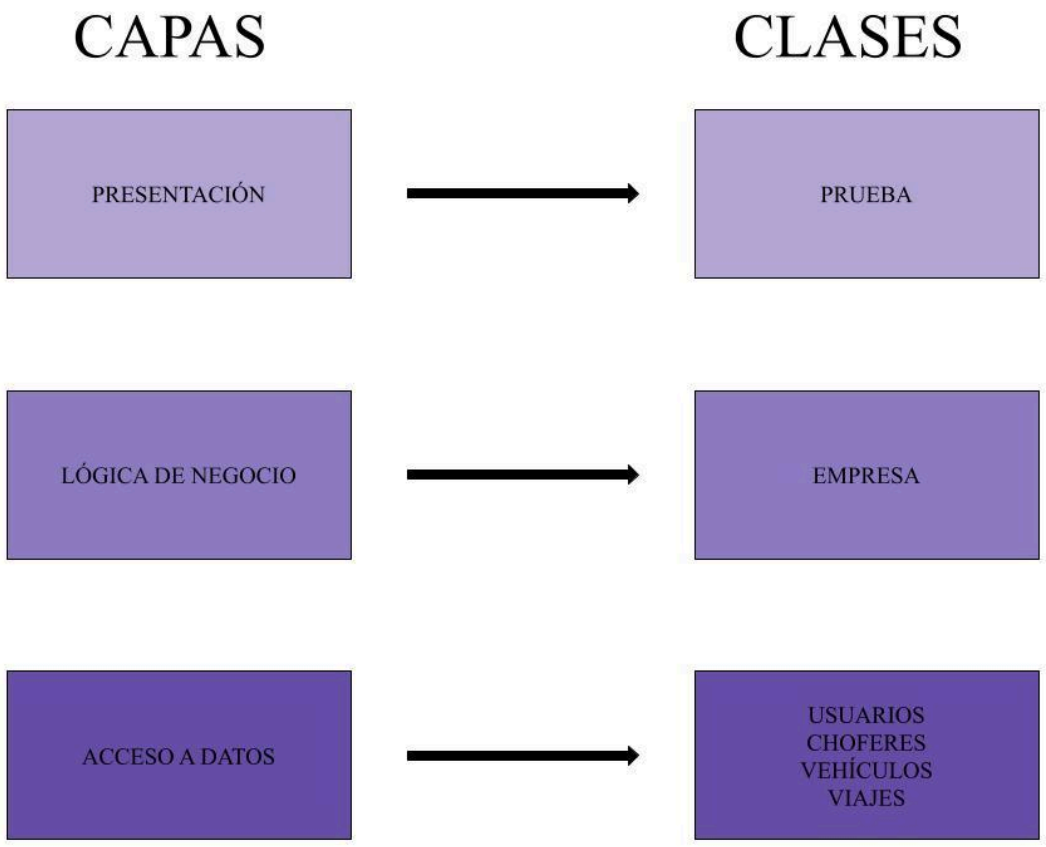
perfeccionar nuestra forma de trabajar en grupo.

DESARROLLO

MVC

Uno de los desafíos para esta segunda parte del trabajo práctico fue la de adaptar la arquitectura de 3 capas modelo-lógica de negocio-presentación a una arquitectura del tipo modelo-vista-controlador.

En la primer etapa, nuestro modelo de 3 capas estaba representado a nivel clases de la siguiente manera



Sin embargo, para adaptar esta disposición a un modelo MVC, las clases debieron reacomodarse en nuevas categorías. De esta manera, el diagrama para nuestra aplicación quedó conformado de la siguiente manera:

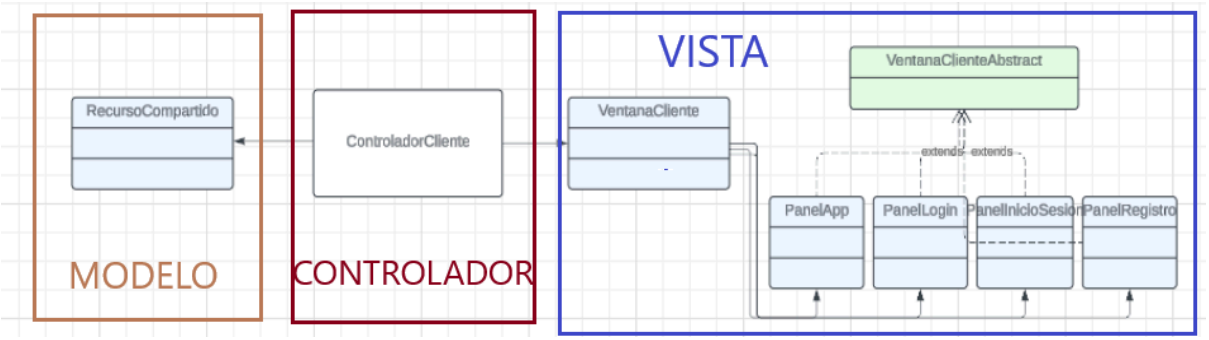


Gráfico 1 - Diagrama MVC aplicación

En la parte izquierda del gráfico (1) podemos observar una nueva clase llamada recurso compartido. Como uno de los ejes centrales de esta segunda parte fue el tratamiento de la concurrencia, se decidió crear esta nueva clase para evitar sobrecargar nuestra clase Empresa ,utilizada con patrón Singleton y Facade durante la primera etapa, encargada de ejecutar la lógica de negocio. La misma será explicada a detalle en secciones posteriores.

Además, como el propósito también es brindar una experiencia de usuario, surge el desarrollo de clases destinadas a representar la interfaz gráfica. De esta forma, nuestra clase prueba que representaba a la presentación y era un simple main hardcodeado, ahora se convirtió en una serie de clases cuyo propósito es brindar un espacio interactivo visual al cliente

Modelo

Si quisiéramos hacer una descripción más detallada de qué lugar ocupan nuestras antiguas clases de modelo (Cliente,Chofer etc) actualmente podemos diagramar lo siguiente:

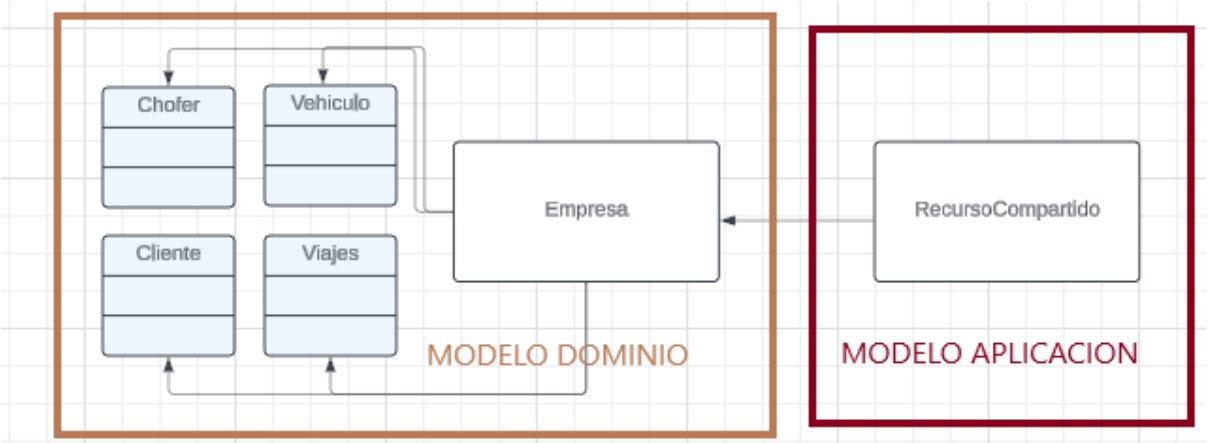


Gráfico 2 - Modelo del patrón MVC

Como puede observarse, las clases que durante la primera parte conformaron el modelo aún siguen siendo parte del mismo, con la inclusión ahora de la clase Empresa. Específicamente estas clases pasaron a formar parte del modelo del dominio, puesto que representan el conocimiento y la lógica de negocio central de la aplicación. Se enfocan en los datos y comportamientos propios del problema que la aplicación está tratando de resolver.

Por su parte, la clase ***RecursoCompartido*** representa el modelo de la aplicación que proporciona operaciones directamente relacionadas con la interacción del usuario y la interfaz gráfica, y al mismo tiempo cuenta con los medios para comunicar a la vista de los cambios que sucedan en el modelo del dominio. En nuestro caso particular, esa comunicación se hace abstractamente por medio del empleo del patrón Observer-Observable, como se verá más adelante.

Vista

Por otro lado, en lo que respecta a la implementación de la vista de nuestra aplicación, cabe hacer alguna aclaraciones del porque de la disposición de jerarquías mostrada en el gráfico **(3)**

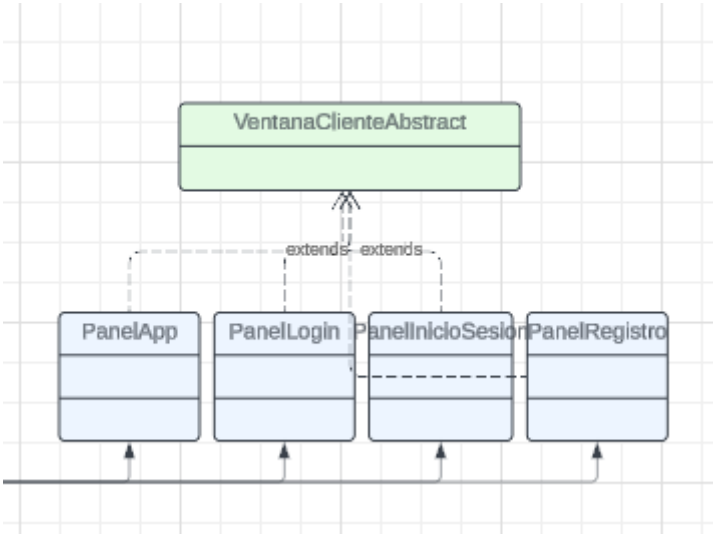


Gráfico 3 - Vista del MVC

Si bien la primera opción que surge a la hora de nuclear comportamiento común entre distintas clases, es la implementación de interfaces, optamos por un camino distinto. Una de nuestras intenciones era poder brindar una interfaz agradable al usuario, tanto estética como funcionalmente. Sin embargo, nos encontramos con limitaciones en los componentes estándares presentes en la librería gráfica Swing.

Una de las características que buscábamos era la opción de tener fondos personalizados y adaptables para nuestras ventanas. Ante la incapacidad de las interfaces de poder brindar implementaciones a los cuerpos de los métodos, nos encontramos con la opción de implementar una clase abstracta de la cual heredaran las distintas ventanas que conforman nuestra aplicación. Esto permitió que en la clase ***VentanaClienteAbstract*** se implemente un método paint() tal que nos permitiera poner el fondo que desearamos al panel que vayamos a utilizar.

Además, hicimos que implementara algunas interfaces útiles a la hora de validar datos de la vista, como lo son *MouseListene*r o *KeyListener*. De ese modo, la clase abstracta definio un comportamiento común para establecer el fondo de cada ventana y permitió que las clases implementen métodos de las interfaces recién mencionadas solo dándole cuerpo a los que ellas necesitaran para llevar adelante su funcionamiento.

Por último, el hecho de que todas nuestras ventanas de la aplicación compartieran un ancestro en la línea de herencia, nos permitio valernos del polimorfismo para ejecutar las transiciones entre ventanas de login,registro, inicio de sesión y la ventana donde se piden viajes

Nuestra ventana de aplicación funciona de la siguiente manera:

Una instancia de la clase ***VentanaCliente*** extiende de ***JFrame*** y es aquella a la cual contiene como atributos a los paneles que representan cada ventana (el de login, registro etc).

Todos ellos son variables de tipo ***VentanaAbstractCliente,*** y cada una cuenta con botones y campos distintos (por ejemplo en la ventana de login se puede rellenar tu nombre y contraseña y en la de pedir viaje el formulario).

Por tal motivo, ***VentanaAbstractCliente*** define los métodos generales para trabajar con todas las ventanas diferentes (getNombre() para registro, getCantPax() para el formulario) y luego cada una de sus extendidas implementó los que corresponden.

De esta forma, ***VentanaCliente*** actúa como clase contenedora principal y coordina las transiciones entre sus paneles de la siguiente manera

```
 */
private void switchPanel(JPanel ventana) {
    getContentPane().removeAll();
    getContentPane().add(ventana);
    currentVentana= (VentanaClienteAbstract) ventana;
    revalidate();
    repaint();
}
```

Fragmento 1 - transiciones de ventanas

El método adjunto en el fragmento 1, recibe por parámetro la ventana próxima, remueve el panel actual de ventana cliente, coloca el nuevo y vuelve a graficarse. Esto permite, por ejemplo, que al apretar en iniciar sesión, y que dicha operación resulte exitosa, la ventana haga la transición de la de inicio de sesión a la de formulario de viaje.

La variable ***currentVentana*** es una referencia de tipo ***VentanaClienteAbstract*** que mantiene ***VentanaCliente*** a la ventana actual. De este modo, cuando requiera información para pasarle al controlador o para validar datos de entrada, ***VentanaCliente*** lo hace sin distinguir si la ventana actual es la de login o la de formulario de viaje llamando simplemente al método correspondiente de ***currentVentana***

Controlador

Por último, referente al diseño MVC, nos queda por explicar la implementación del controlador.

Para controlar la vista del cliente que usa la app, recién comentada en la sección anterior, se desarrolló una clase ***ControladorCliente***.

Gráfico 4 - Controlador MVC

Como puede observarse en el gráfico **(4)**, la clase ***ControladorCliente*** hace de nexo entre la ventana del cliente y el recurso compartido, trasladando las peticiones del usuario que interactúa con la interfaz gráfica hacia el recurso compartido que tiene la capacidad de atender dichas solicitudes.

Además, ni el recurso compartido ni la vista conocen directamente al controlador, pero el si mantiene referencias activa a ambos.

Para coordinar el flujo de la aplicación en base a las interacciones del cliente con los menús y botones, el comportamiento central del controlador nace de aplicar un patrón de comandos. Es decir, nuestro controlador implementa la interfaz ***ActionListener***.Al crearse se le pasa como parámetros de constructor una ***VentanaCliente***, que representa la vista que controlara, y el recurso compartido, que es su medio de comunicación con el modelo.

De esta manera, se setea en los componentes de la ventana del cliente (botones sobre todo) al controlador como ***ActionListener***. Esto permite que cuando el usuario interactúe con la ventana apretando dichos botones, estos disparen un ***ActionEven***t que va directo al método *ActionPerformed()* de la clase controladora, donde se decide qué acción llevar adelante en base al comando recibido

```
@Override
public void actionPerformed(ActionEvent evento) {
    switch(evento.getActionCommand()) {
        case "ATRAS REG":
        case "ATRAS SESION":
            vista.setLogin();
            break;

        case "REGISTRAR":
            vista.setRegistro();
            break;

        case "INICIAR SESION":
            vista.setInicioSesion();
            break;

        case "INTENTO REGISTRO":
            intentoRegistro();
            break;

        case "INTENTO INICIO SESION":
            intentoInicioSesion();
            break;

        case "PAGAR":
            vista.setDialogFinViaje();
            break;
```

Fragmento 2 - Cómo actúa el controlador

Por lo tanto, cada botón de las distintas ventanas tiene un action command asociado. Una vez el usuario aprieta uno de ellos la solicitud es recibida por el controlador y según el resultado del *switch()* se ejecuta el método que corresponde, que generalmente involucra alguna interacción con el modelo.

Ahora sí, de esta forma damos cierre a la explicación de la implementación del diseño MVC. Sin embargo, cabe hacer una última aclaración. La explicación se dio en base a la ventana del cliente puesto que es la más compleja y la que cuenta con mayor interacción humana.

Es importante recalcar que también forman parte de nuestra vista la clase ***VentanaGeneral,*** que simplemente muestra por pantalla sucesos asociados a la simulación y por ende no cuenta con controlador, y la ventana inicial donde se ajustan los parámetros de la simulación. Dicha ventana si cuenta con un controlador, pero las acciones del mismo se reducen simplemente a 2: iniciar simulación nueva con parámetros cargados por el usuario o iniciar simulación con datos persistidos

PATRONES Y DISEÑO

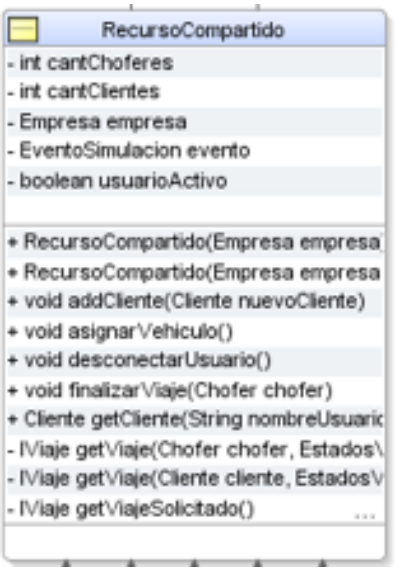
Concurrencia

La parte de concurrencia en este trabajo es llevada a cabo por las clases que extienden ***Thread*** (***ClienteThread***, ***ChoferThread*** y ***SimulacionThread***), la clase ***RecursoCompartido*** y la clase ***Simulacion***. El problema de concurrencia surge debido a que cada hilo trata de acceder a recursos de la empresa al mismo tiempo; el cliente trata de pedir viajes y posteriormente pagarlos; los choferes tratan de tomarlos y posteriormente finalizarlos; el sistema trata de asignar vehículos a los viajes. Como el acceso múltiple a un mismo recurso puede generar problemas (inconsistencia o pérdida de datos) surge la necesidad de una clase (***RecursoCompartido***) que se encargue de definir un orden y de limitar quién interactúa y en qué momento con los recursos y la empresa.

RecursoCompartido

El recurso compartido como ya fue mencionado es la clase a través de la cual los hilos van a comunicarse con la empresa. Tiene la responsabilidad de comunicar los hilos con la empresa y restringir el uso de los recursos de la misma para que solo un hilo simultáneo pueda acceder. En términos de la materia podemos decir que la clase ***RecursoCompartido*** actúa como un monitor.

La clase ***RecursoCompartido*** tiene los siguientes atributos y métodos.



UML 1 - Listado de métodos y atributos de la clase RecursoCompartido

Utiliza el atributo ***Empresa*** para guardar una referencia a la empresa y así poder extender los pedidos de los hilos a esta. Los atributos ***cantChoferes*** y ***cantClientes*** son los

que corresponden a la configuración de la simulación en la vista. El atributo evento que es de la clase ***EventoSimulacion*** será usado en los eventos que son de interés para los observadores para enviarlo a través de notifyObservers(). No hay problema con que sea un atributo único para el recurso compartido, ya que cada vez que se modifica este es un método synchronized por lo que solo un hilo a la vez tiene acceso.

Los métodos que se van a explicar a continuación son los referidos al manejo del sincronismo, también están los métodos relacionados a consultas a la empresa como ***hayViajesPagos*** o ***hayViajesSolicitados*** pero no son implementados en el recurso compartido por lo que no se hará énfasis en ellos.

pedirViaje

```
public synchronized void pedirViaje(Cliente cliente,String zona, int mascota, String tipoServicio, int equipaje, int cantPax, double distancia, Loc
        throws ExceptionPedido, ExceptionVehiculoDisp
{

   if(cantChoferes>0)
   {
    try{
        empresa.pedirViaje(cliente.getNombreUsuario(), zona, mascota, tipoServicio, equipaje, cantPax, distancia, fecha);
        evento = new EventoSimulacion("solicito viaje y espera",cliente,null,null,TipoEvento.CLIENTE);
    }
    catch(ExceptionVehiculoDisp e){
        evento = new EventoSimulacion("realizo pedido pero no habia auto que cumpla las especificaciones",cliente,null,null,TipoEvento.CLIENTE);
        setChanged();
        notifyObservers(evento);
        notifyAll();
        throw  e;
    }
    catch(ExceptionPedido e){
       evento = new EventoSimulacion("realizo pedido pero fue rechazado "+e.getMessage().toLowerCase(),cliente,null,null,TipoEvento.CLIENTE);
       setChanged();
       notifyObservers(evento);
       notifyAll();
       throw  e;
    } catch (ExceptionUsuario ex) {

    }
   }else
        evento = new EventoSimulacion("no pudo realizar pedido porque no hay choferes disponibles ",cliente,null,null,TipoEvento.CLIENTE);


   setChanged();
   notifyObservers(evento);
   notifyAll();
}
```

Fragmento 3 - Parte inicial de pedirViaje en el recurso compartido.

El método pedirViaje recibe los parámetros necesarios para solicitarle a la empresa un viaje, en caso de que los datos no sean válidos se ataja la excepción, se avisa a los observadores y se notifica a los hilos que están tratando de acceder al recurso que el mismo está libre. La condición para ejecutarse es que todavía queden hilos choferes activos.

asignarVehiculo

```
while(simulacionIsActiva() && viajeSolicitado == null)
{
    try {
         evento = new EventoSimulacion("El sistema intento asignar auto pero no hay viajes solicitados", null, null, null, TipoEvento.SISTEMA);
         setChanged();
         notifyObservers(evento);
         wait();
```

Fragmento 4 - Primer ciclo del método.

```
   while(simulacionIsActiva()  && !empresa.asignarVehiculo(viajeSolicitado))
   {
       try {
           evento = new EventoSimulacion("El sistema intento asignar vehiculo al viaje del cliente " + viajeSolicitado.getCliente().getNombreUsuario()+" per
                                   viajeSolicitado.getCliente(), null, null, TipoEvento.SISTEMA);
           setChanged();
           notifyObservers(evento);
           wait();
        } catch (InterruptedException ex) {}
       viajeSolicitado = getViajeSolicitado();
   }
   if(viajeSolicitado != null)
       evento = new EventoSimulacion("El sistema asignó "+viajeSolicitado.getVehiculo().getTipo()+" al viaje del cliente " + viajeSolicitado.getCliente().ge
                                   viajeSolicitado.getCliente(), null, null, TipoEvento.SISTEMA);
   else
       evento = new EventoSimulacion("El sistema se apaga",null,null,null,TipoEvento.SISTEMA);

   setChanged();
   notifyObservers(evento);
   notifyAll();
```

Fragmento 5 - Segundo ciclo del método.

El método ***asignarVehiculo()*** consta de dos ciclos principales, en el primero de ambos (*fragmento 4.*), el sistema trata de encontrar un viaje en estado solicitado, sí no es capaz de encontrarlo (ya que no hay pedidos pendientes en la empresa), se genera un nuevo evento para notificar a los observadores y se libera el uso del recurso compartido a través de *wait()*.

Suponiendo que hubiese un viaje en estado solicitado, el sistema entonces entrará en el segundo ciclo (*fragmento 5.*) el cual se encargará de tratar de asignarle un vehículo al último viaje solicitado. De no haber ningún vehículo **disponible** (el o los que cumplen las especificaciones para el viaje están siendo utilizados en otros) se genera un nuevo evento para notificar a los observadores y se libera el uso del recurso compartido a través de *wait()*. En caso de que sí se logre asignar un vehículo a algún viaje en estado solicitado se sale del ciclo de y se genera un evento nuevo donde se comunique esto a los observadores, para posteriormente liberar el recurso compartido a través de *notifyAll()*.

Otra forma de que este método llegue a su fin es que la simulación termine, lo cual es considerado como condición de corte de ambos ciclos y al final es notificado a los observadores a través de un evento.

tomarViaje:

```
while(hayClientes() && !hayViajeConVehiculo() && (cantViajesPedientes > 0 || usuarioActivo))
{
    try {
        wait();
    } catch (InterruptedException ex) {}
}
```

Fragmento 6 - Ciclo principal con wait() del método.

```
if(hayClientes())
{
    empresa.asignarChofer(chofer);
    viaje = getViaje(chofer,EstadosViajes.INICIADO);
    if(viaje!=null)
        evento =  new EventoSimulacion("tomo el viaje del cliente "+viaje.getCliente().getNombreUsuario(),viaje.getCliente(),chofer,null,TipoEvento.CHOFE
    else
        evento =  new EventoSimulacion("intento tomar viaje pero no habia ninguno con vehiculo asignado",null,chofer,null,TipoEvento.CHOFER);
}
else
    evento =  new EventoSimulacion("descubre que no hay mas clientes se retira de la empresa",null,chofer,null,TipoEvento.CHOFER);

setChanged();
notifyObservers(evento);
notifyAll();
}
```

Fragmento 7 - En caso de salir del ciclo principal.

El método ***tomarViaje()*** recibe como parámetro el chofer que quiere ser asignado a un viaje, esto resulta de la necesidad de que cada hilo de chofer diferencie su llamada método y posteriormente pueda ser asignado a un viaje, también recibe la cantidad de viajes pendientes para poder usarla en el ciclo principal del ***wait()***. El ciclo principal (*fragmento 6.*) de este método synchronized tiene como condición de corte que no haya más clientes, que se haya encontrado un viaje con vehículo asignado o que el usuario esté inactivo y el chofer no tenga más viajes.

Al salir de dicho ciclo (fragmento 7) se evalúa que aún queden clientes activos. En caso afirmativo, se solicita a la empresa que asigne el chofer a un viaje con vehículo, caso contrario el chofer se retira de la empresa. En ambas situaciones se genera un evento que se comunica a los observadores describiendo lo sucedido y se libera el uso del recurso compartido con *notifyAll()*.

pagarViaje:

```
while(cantChoferes>0 && !viajeIniciado(cliente))
{
    try {
        wait();
    } catch (InterruptedException ex) {
```

Fragmento 8 - Ciclo principal.

```
if(viajeIniciado(cliente))
{
    try {
         empresa.pagarViaje(cliente);
         evento = new EventoSimulacion("pago el viaje y se retiro del vehiculo",
                 cliente,getViaje(cliente,EstadosViajes.PAGO).getChofer(),null,TipoEvento.CLIENTE);
    } catch (ExceptionSinViajeaPagar ex) {
        //no entramos nunca porque validamos q tenga viaje iniciado
    } catch (ExceptionUsuario ex) {

     }
}else
       evento = new EventoSimulacion("cancela el viaje porque no hay choferes disponibles",
               cliente,null,null,TipoEvento.CLIENTE);

    setChanged();
    notifyObservers(evento);
}
notifyAll();
```

Fragmento 9 - En caso de salir del ciclo principal

El método ***pagarViaje()*** recibe como parámetro al cliente de forma que cada hilo cliente que esté buscando pagar su viaje pueda diferenciarse del resto al momento de obtener el control del recurso compartido.

El ciclo principal (*fragmento 8.*) tiene como condición de corte que el cliente tenga un viaje iniciado o que no haya más hilos de chofer activos, de no ser así el hilo liberará el uso del recurso compartido a través de *wait()*.

Cuando se sale del ciclo (*fragmento 9.*) se revisa sí salió porque encontró un viaje iniciado, caso contrario la simulación terminó. En caso de que sea porque se encontró un viaje iniciado se llama al método de la empresa para marcar el viaje como pagado, sino significa que no hay más choferes activos y se “cancela” el viaje. En todos los casos se genera un evento que se comunica a los observadores y posteriormente se libera el uso del recurso compartido con ***notifyAll()***.

finalizarViaje:

```
public synchronized void finalizarViaje(Chofer chofer, int cantViajesPendientes)
{

    while (simulacionIsActiva() && !viajePago(chofer) && (cantViajesPendientes > 0 || usuarioActivo))
        try {
            wait();
        } catch (InterruptedException ex) {}


    if (viajePago(chofer))
    {
        try {
            empresa.finalizarViaje(chofer);
            evento = new EventoSimulacion("finalizo el viaje y devolvio el vehiculo",
                    getViaje(chofer, EstadosViajes.FINALIZADO).getCliente(), chofer, null, TipoEvento.CHOFER);
            setChanged();
            notifyObservers(evento);
        }
        catch (ExceptionChofer ex) {}
        catch (ExceptionChoferSinViajesPagos ex) {}
    }
    notifyAll();

}
```

Fragmento 10 - Método finalizarViaje.

El último método synchronized referido al flujo de los viajes es ***finalizarViaje()***, el cual recibe como parámetro el chofer de forma que cada hilo de chofer que acceda al método asegure terminar un viaje propio y además recibe la cantidad de viajes pendientes de ese hilo, para usarlo de condición en el while del ***wait()***. De forma similar a los métodos anteriores, en la condición se considera el fin de la simulación, la existencia de un viaje en estado pago y que no tenga más viajes pendientes y el usuario se haya desconectado. Suponiendo que se encuentra un viaje en estado pago se lo finaliza, se crea un evento que describe esto y se lo comunica a los observadores. En ambos casos de corte (simulación finalizada y viaje pago encontrado) se termina con un *notifyAll()* liberando el recurso compartido.

subChofer y ***subCliente:***

Por último, se cuenta con los métodos ***subChofer()*** y ***subCliente()*** que se encargan de restar uno a la cantidad de choferes y clientes activos respectivamente. Estos métodos deben ser synchronized ya que afectan atributos y pueden ser llamados por múltiples hilos a la vez, por lo que haciéndolos synchronized es seguro el hecho de que solo un hilo sea de cliente o chofer, pueda afectarlos. Es necesario que esto ocurra puesto que dos de las condiciones de finalización de la simulación está asociada a la terminación de todos los hilos chofer o cliente de la simulación, y es el recurso compartido quien lleva cuenta de ello.

Threads

En el trabajo los hilos son utilizados para la ejecución en simultáneo de los bots de clientes (***ClienteThread***), choferes (***ChoferThread***) y el sistema (***SistemaThread***). Cada uno

de estos tienen la responsabilidad que se estableció para los mismos en el enunciado del TP.

Todos los hilos, reciben como parámetro de su constructor el recurso compartido con el fin de que puedan acceder e interactuar con este. Por su parte, el ***ClienteThread*** recibe al cliente que representa, y de la misma forma el ***ChoferThread*** recibe su chofer. Podemos ver en los métodos run de cada uno como se sigue lo que se muestra en el cuadro del enunciado

```
@Override
public void run()
{
    while(cantViajes>0 && rc.getCantChoferes()>0)
    {

       try{
             Util.espera();
             rc.pedirViaje(cliente, zonas[Util.rand(4)], Util.rand(0,1), "Transporte",
                      Util.rand(0,1),Util.rand(1,12),Util.rand(4000), LocalDateTime.now());

             Util.espera();
             rc.pagarViaje(cliente);
             this.cantViajes--;

       }catch(Exception e){}
     }
    rc.subCliente(cliente);
}
```

Fragmento 11 - Método run de ClienteThread.

El hilo del cliente busca acceder al recurso compartido, primero, para solicitar un viaje y posterior a eso para pagarlo. Dentro del método de pedir viaje, se asignan valores aleatorios válidos y no válidos para simular la probabilidad de que los clientes se equivoquen en los valores que ingresan.Como ya fue mencionado el recurso compartido actúa como un monitor, por lo que por más de que todos los hilos de cliente traten de acceder al mismo tiempo al recurso compartido para ejecutar esto, solamente uno de ellos podrá acceder y realizar la acción que desea.

Suponiendo que un cliente logre solicitar el viaje, tratará de pasar al método de pagar y quedará ahí hasta que cumpla con las condiciones para pagar el viaje. Podemos hacerlo de esta manera sin comprobar nada dentro de este ciclo ya que el recurso compartido se encarga de gestionar estas interacciones del cliente con la empresa.

La condición de corte del método *run()* de los clientes, es que no haya más choferes activos o que hayan agotado su cantidad de viajes.

```
@Override
public void run()
{
    while(rc.getUsuarioActivo() || (cantViajes>0 && rc.hayClientes()))
    {
        Util.espera();
        rc.tomarViaje(chofer, cantViajes);

        Util.espera();
        rc.finalizarViaje(chofer, cantViajes);
        cantViajes--;
    }
    rc.subChofer(chofer);
}
```

Fragmento 12 - Método run de ChoferThread.

El hilo del chofer actúa de forma muy similar al del cliente, solo que con el objetivo de tomar un viaje y luego de finalizarlo. Otra cosa que realizan igual tanto el chofer como el cliente es una vez finalizados los viajes que deseaban realizar, llaman a los métodos ***subChofer*** y ***subCliente*** para avisar al recurso compartido la muerte de ese hilo.

Sin embargo, el método run del chofer requiere de una condición de corte especial. No solo debe contemplar que no se agoten sus viajes, sino que también existe la posibilidad de que el usuario desde la ventana de la app solicite viaje y, por ende, deben existir mientras el usuario siga activo en la ventana de la app, choferes que puedan atenderlo. Por esa razón, la condición del while es un **or**: finaliza cuando acabe sus viajes y el usuario de la app esté desconectado o, si se quedo sin viajes y todavía hay clientes, continúa hasta que se desconecte el usuario.

```
@Override
public void run()
{
    while(rc.simulacionIsActiva())
    {
        Util.espera();
        rc.asignarVehiculo();
    }
}
```

Fragmento 13 - Método run de SistemaThread.

El sistema thread resulta ser el más simple de los tres ya que está activo siempre que la simulación está activa y su único propósito es el de asignar vehículos a los viajes.

Simulación

La clase simulación es la encargada de unir todo lo que se ha descrito hasta ahora. Tiene como atributos un arreglo de ***ChoferThread***, un arreglo de ***ClienteThread*** y al recurso compartido. Simulación se instancia una única vez en el main del programa para ser pasada como parámetro al controlador del menú simulación que la utiliza a través de los siguientes métodos dependiendo de los botones con los que interactúe el usuario en la ventana de la simulación.

El método que inicializa la simulación recibe un único parámetro ***p*** de la clase ***ParametrosSimulacion*** cuya función es la de almacenar todas las configuraciones ingresadas a través de la vista donde se configura la simulación, con esto, se inicia la simulación de la siguiente forma.

```
public void iniciarSimulacionNueva(ParametrosSimulacion p)
{
     Empresa empresa = Empresa.getInstance();
     rc =  new RecursoCompartido(empresa,p.getCantClientes(),p.CantChoferes());

     initClientes(empresa,p.getCantClientes(), p.getCantMaxViajeCliente());

     initChoferes(empresa,p.getCantChoferTemporario(), p.getCantChoferContratado(),
             p.getCantChoferPermanente(), p.getCantMaxViajeChofer());

     initVehiculos(empresa,p.getCantAutos(), p.getCantMotos(), p.getCantCombis());
     simular(empresa,p);
}
```

Fragmento 14 - Método iniciarSimulacionNueva.

Se crea el recurso compartido que utilizará la simulación, se crean los hilos de los clientes y choferes, el hilo del sistema y se llama al método simular.

Simular

```
try {
     persisteEmpresa.abrirOutput("Empresa.xml");
     EmpresaDTO empresaDTO = UTILEmpresa.empresaDTOFromEmpresa(empresa);
     persisteEmpresa.escribir(empresaDTO);
     persisteEmpresa.cerrarOutput();
     if(parametros!=null)
     {
          persisteParametros.abrirOutput("Parametros.xml");
          persisteParametros.escribir(parametros);
          persisteParametros.cerrarOutput();
     }
} catch (IOException ex) {
     Logger.getLogger(Simulacion.class.getName()).log(Level.SEVERE, null, ex);
}


for(ClienteThread c : robotsCliente)
      c.start();

 SistemaThread sistema = new SistemaThread(rc);
 sistema.start();

for (ChoferThread c : robotsChofer)
      c.start();
```

Fragmento 15 - Persistencia dentro del método e inicio de hilos.

Este método se encarga de crear los observers que avisarán a las vistas de los cambios producidos en el recurso compartido. Crea el controlador del cliente, el cual recibe a la vista del cliente y al recurso compartido. También se encarga de persistir los datos previo a la inicialización de la simulación, de esta forma la siguiente vez que se quiera realizar una simulación se podrá iniciar con estos datos que se persisten en este momento. Por último se utiliza el método *start()* en cada uno de los hilos.

IniciarSimulacionConDatosViejos

```
public void iniciarSimulacionConDatosViejos(ParametrosSimulacion parametros)
{
    PersistenciaXML leeEmpresa = new PersistenciaXML();

    try {
        leeEmpresa.abrirInput("Empresa.xml");
        EmpresaDTO empresaDTO = (EmpresaDTO) leeEmpresa.leer();
        Empresa empresa = UTILEmpresa.empresaFromEmpresaDTO(empresaDTO);
        leeEmpresa.cerrarInput();


        rc = new RecursoCompartido(empresa, empresa.getChoferLista().size(), empresa.getUsuarioLista().size()-1);
        initThreads(empresa,parametros.getCantMaxViajeChofer(),parametros.getCantMaxViajeCliente());
        simular(empresa,null);

    } catch (IOException ex) {
        Logger.getLogger(Simulacion.class.getName()).log(Level.SEVERE, null, ex);
    }catch (ClassNotFoundException ex) {
        Logger.getLogger(Simulacion.class.getName()).log(Level.SEVERE, null, ex);
    }
}
```

Fragmento 16 - Método de simulación con los datos persistidos previamente.

Este método a diferencia del anterior carga los datos persistidos previamente y genera la empresa como estaba con las configuraciones anteriores. Con los datos de la persistencia de la empresa y los parámetros de la simulación que también fueron persistidos previamente y pasados como parámetro a este método comienza a simular con la misma configuración inicial que se había persistido.

Patrón Observer / Observable

En este proyecto utilizamos el patrón observer / observable para poder comunicar cambios en el recurso compartido utilizado en concurrencia a las vistas.Todas las ventanas que muestran texto relacionado con el flujo de un viaje funcionan a través de las tres clases de observadores que se definieron ***OjoGeneral***, ***OjoChoferSimulacion***, ***OjoCliente***, ***OjoChoferSimulacion***, las cuales todas extienden de una clase ***OjoAbstracto*** que implementa ***Observer*** y define un atributo observado, el cual (en el caso de nuestra solución) resulta ser para todos los observadores, el recurso compartido.

Este patrón resultó de mucha ayuda a la hora de manejar múltiples ventanas que reciben su información de una misma fuente, ya que simplifica la llamada a mostrar por pantalla a dos métodos *setChanged()* y *notifyObservers(…)*, además desliga al recurso compartido de interactuar directamente con la vista lo cual no entra en sus responsabilidades, y de enterarse con cuántos receptores (observadores) está interactuando por lo que colabora con no sobrecargar esta clase y reduce el acoplamiento.

Para las clases se va a mostrar solo dos métodos update representativos, y un constructor ya que el constructor es similar en todos.

```
public OjoGeneral(VentanaGeneral vista, Observable observado)
{
    this.vista = vista;
    this.observado = observado;
    observado.addObserver(this);
}
```

Fragmento 17 - Constructor de OjoGeneral.

En el constructor se define el objeto que va a ser observado por el observer, se agrega a sí mismo a la lista de observadores que tiene el objeto que observa y define, en caso de que lo necesite, a algún Cliente o Chofer particular para filtrar dentro de los datos que observa.

OjoGeneral

```
@Override
public void update(Observable o, Object e) {
    super.update(o, e);

    EventoSimulacion evento;
    evento = (EventoSimulacion)e;

       switch(evento.getTipo())
       {

           case CLIENTE:  vista.appendGeneral(evento.getCliente().getNombreUsuario()+" "+evento.getMensaje()+"\n");
           break;
           case CHOFER:  vista.appendGeneral(evento.getChofer().getNombre()+" "+evento.getMensaje()+"\n");
           break;
           case SISTEMA: vista.appendGeneral(evento.getMensaje()+"\n");

       }


}
```

Fragmento 18 - Método update de OjoGeneral.

Lo primero que hace el ojo general es llamar a *super.update(o, e)* principalmente para comprobar que el objeto es el objeto observado, una vez tengo esto asegurado, puedo proseguir. El ojo general se encarga de mostrar los datos relacionados con todos los viajes que están sucediendo. Cada vez que hay un evento nuevo siendo informado por el recurso compartido el ojo general es el encargado de tomar los valores de la clase ***EventoSimulacion*** definida y mostrar por la pantalla general de la simulación estos datos. Para ello usa el atributo tipo dentro del evento, de está forma dependiendo el tipo de cambio que hubo en el recurso compartido es entonces la forma en que debe mostrar el mensaje.

OjoClienteSimulacion

```
@Override
public void update(Observable o, Object e) {
    super.update(o, e);

    EventoSimulacion evento = (EventoSimulacion)e;

    Cliente clienteEvento = evento.getCliente();
    if(clienteEvento != null && clienteEvento.equals(cliente))
    {
        switch(evento.getTipo())
        {
           case CLIENTE:  vista.appendCliente(evento.getCliente().getNombreUsuario()+" "+evento.getMensaje()+"\n");
           break;
           case CHOFER:  vista.appendCliente(evento.getChofer().getNombre()+" "+evento.getMensaje()+"\n");
           break;
           case SISTEMA: vista.appendCliente(evento.getMensaje()+"\n");
        }
    }
}
```

Fragmento 19 - Método update de OjoClienteSimulacion.

La diferencia más importante con la clase ***OjoGeneral*** está en la condición para mostrar algo en la vista, que es que primero debe suceder que el evento que llegó al observador sea un evento que tenga un cliente no nulo y además sea el cliente que está observando, es de la misma forma que se filtran los mensajes que tienen que llegar a cada vista en los otros observadores.

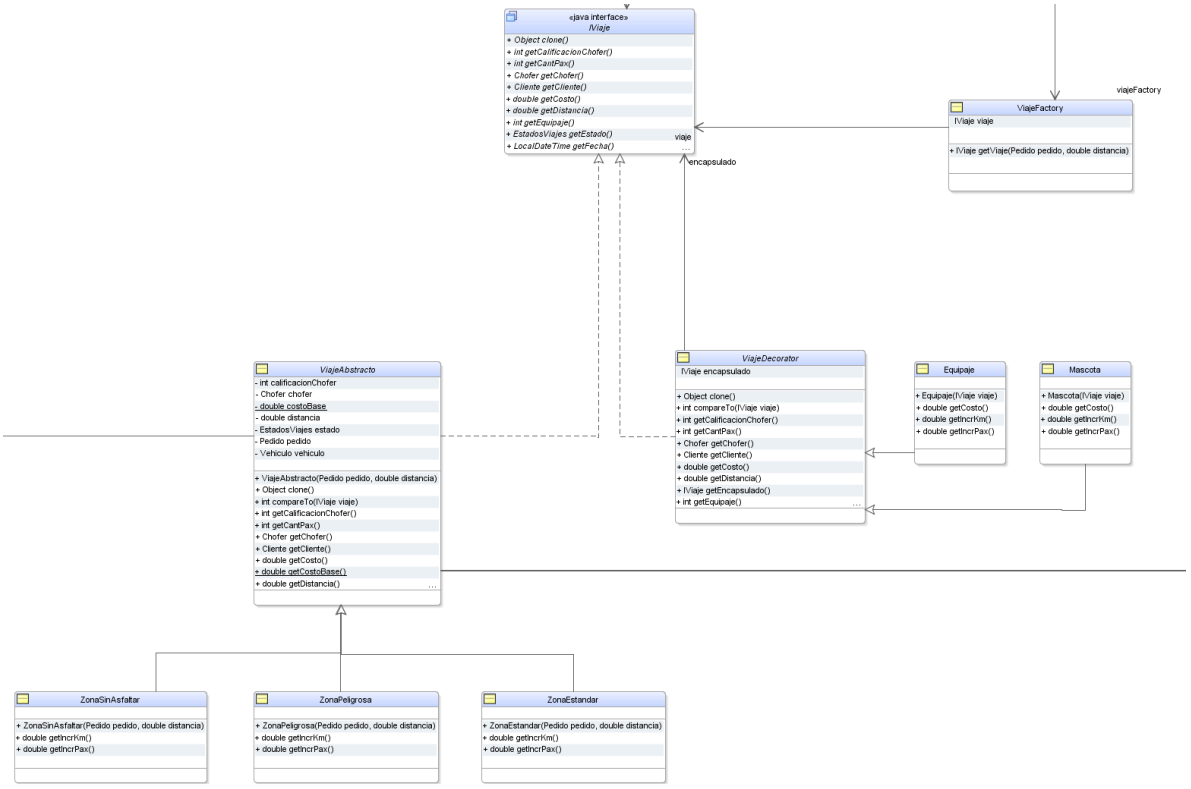
Así nos aseguramos que sólo se muestre lo que deseamos en cada vista y además el recurso compartido no está al tanto de ninguna de estas cosas ya que no es su responsabilidad.

Patrón Decorator

Otro de los patrones que resultó de mucha utilidad en el trabajo fue el patrón decorator. El mismo nos brindó la alternativa flexible a la hora de extender la funcionalidad. En particular, la situación que derivó en el empleo de este método, fue el cálculo de costo de los viajes. El mismo depende de muchos factores diferentes, tales como la zona, cantidad de pasajeros, si lleva equipaje y mascota. Para evitar escribir métodos sobrecargados de ifs que chequearan todos los atributos en cada una de posibilidades para poder calcular el costo, optamos por modificar este comportamiento, decorando únicamente los métodos que fueran necesarios.

Entre los componentes fundamentales que participan en el funcionamiento de esta clase y, que representan o almacenan los datos con los que trabaja dicho subsistema, se encuentran:

- ***IViaje*** interfaz que nuclea el comportamiento de los viajes.
- ***ViajeAbstracto*** padre de clases concretas ***ZonaEstandar***, ***ZonaPeligrosa*** y ***ZonaSinAsfaltar***.
- ***ViajeDecorator*** padre de las clases decoradoras *Equipaje* y *Mascota*.
- ***ViajeFactory*** encargada de la creación de viajes.



Por un lado, ***IViaje*** es la interfaz responsable de modelar el comportamiento de los viajes, predefiniendo qué métodos deben tener. Además, cumple el rol de ancestro común entre ***ViajeAbstracto*** y ***ViajeDecorator***, permitiendo así que al aplicar el patrón Decorator, tanto un viaje decorado como uno no decorado puedan actuar polimórficamente y ser tratados de la misma manera..

Por otro lado, ***ViajeAbstracto*** implementa ***IViaje*** para dar cuerpo a los métodos generales de los viajes, pues es la clase que almacena los atributos del mismo. Ademas, ***ZonaEstandar***, ***ZonaSinAsfaltar*** y ***ZonaPeligrosa*** son las clases concretas extendidas que posteriormente podrán ser encapsuladas en ***ViajeFactory***. Estas clases sobreescriben los métodos *getIncrKm()* y *getIncrPax()* con los valores de incremento que corresponden de acuerdo a la zona y la cantidad de pasajeros y kilómetros recorridos del viaje solicitado.

```
public IViaje getViaje(Pedido pedido, double distancia)
{
    assert pedido != null : "Fallo Pre: El Pedido no puede ser Null ";
    assert distancia > 0 : "Fallo Pre: la distancia debe ser mayor a 0";

    switch(pedido.getZona().toUpperCase()) {
        case "ESTANDAR":
            viaje = new ZonaEstandar(pedido,distancia);
            break;
        case "SIN ASFALTAR":
            viaje = new ZonaSinAsfaltar(pedido,distancia);
            break;
        case "PELIGROSA":
            viaje = new ZonaPeligrosa(pedido,distancia);

    }

    if(pedido.getMascota() != 0) {
        viaje= new Mascota(viaje);
    }

    if(pedido.getEquipaje() != 0) {
        viaje = new Equipaje(viaje);
    }

    return viaje;
}
```

Fragmento 20 - Factory de viajes

Como se ve reflejado en la imagen, ***ViajeFactory*** será la responsable de crear el viaje solicitado en base a los parámetros traspasados por el ***Pedido*** que la ***Empresa*** solicitó y el subsistema de Viajes validó. En dicha creación se instancia primeramente la clase concreta de viaje que corresponda según la zona y luego se encapsula dependiendo de si requiere un vehículo Pet Friendly y/o con baúl.

```
@Override
public double getCosto() {
    return ViajeAbstracto.getCostoBase() * (1 + this.getIncrKm() + this.getIncrPax());
}

/**
 * Obtiene el incremento por pasajero asociado a la opcion Pet Friendly.<br>
 * @return El incremento por pasajero.
 */
public double getIncrPax() {
    return this.getEncapsulado().getIncrPax() + 0.1 * this.getEncapsulado().getCantPax();
}

/**
 * Obtiene el incremento por kilometro asociado a la opcion Pet Friendly.<br>
 * @return El incremento por kilometro.
 */
public double getIncrKm() {
    return this.getEncapsulado().getIncrKm() + 0.2 * this.getEncapsulado().getDistancia();
}
```

Fragmento 21 - Métodos decorados

Cada capa del decorator, es decir, las clases ***Mascota*** y ***Equipaje***, se encargan de decorar los métodos *getIncrKm()* y *getIncrPax()* agregándoles el incremento adicional por la nueva propiedad implementada al incremento anterior del encapsulado por zona, permitiendo de esa forma en *getCosto()* calcular correctamente el valor de acuerdo a los parámetros del pedido particular.

Para ir finalizando esta sección, nos parece interesante comentar que decidimos modelar los diferentes estados posibles de un viaje (***"SOLICITADO","INICIADO","CONVEHICULO","PAGO","FINALIZADO"***) haciendo uso de una de las clases que nos ofrece JAVA, los ***Enums:***

```
/**
 * Vector de Estados que representa los posibles estados de un viaje.
 */
static enum EstadosViajes {
    SOLICITADO, CONVEHICULO, INICIADO, PAGO, FINALIZADO
};
```

Fragmento 22 - Estados de viaje

Además, también aprovechamos la interfaz *Comparable* e hicimos que ***IViaje*** la extienda, de manera tal que tanto viajes decorados como no decorados sean comparables, y ***ViajeAbstracto*** implemente el método *compareTo()*.

```
/**
 * Compara este Viaje con otro viaje segun el costo.<br>
 * @param viaje El otro Viaje con el que se compara.
 * @return El resultado de la comparacion.
 */
public int compareTo(IViaje viaje) {
    return Double.compare(viaje.getCosto(), this.getCosto());
}
```

Fragmento 23 - Uso de interfaz Comparable

La idea de implementar esta interfaz fue darnos una facilidad a la hora de generar el listado de viajes ordenados por costo de mayor a menor

El método que mostramos a continuación, *getListaClonViajesOrdenados()*, lleva adelante la tarea de generar una lista de los viajes ordenada por costos. Para ello, primero recorre la lista de viajes original y clona viaje por viaje. Luego, hace una invocación a ***Colections.sort(),*** método el cual ordena la colección pasada como parámetro usando como criterio de ordenación el valor que devuelve el método *compareTo()* de los objetos que la conforman.

```
private ArrayList<IViaje> getListaClonViajesOrdenados()
{
    ArrayList<IViaje> viajeLista = empresa.getViajeLista();

    ArrayList<IViaje> viajeListaClon = (ArrayList<IViaje>) viajeLista.clone();
    viajeListaClon.clear();

    for(int i = 0; i < viajeLista.size(); i++){
         viajeListaClon.add((IViaje) viajeLista.get(i).clone());
    }

    //Ordena la lista clonada por costo,en IViaje sobreescribi el metodo compare
    Collections.sort(viajeListaClon);

    return viajeListaClon;
}
```

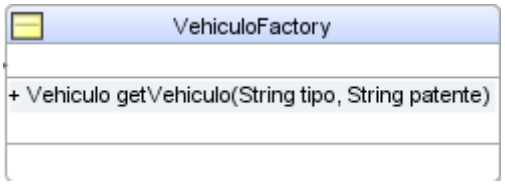
Fragmento 25 - Creación de lista clon de viajes ordenada por costo

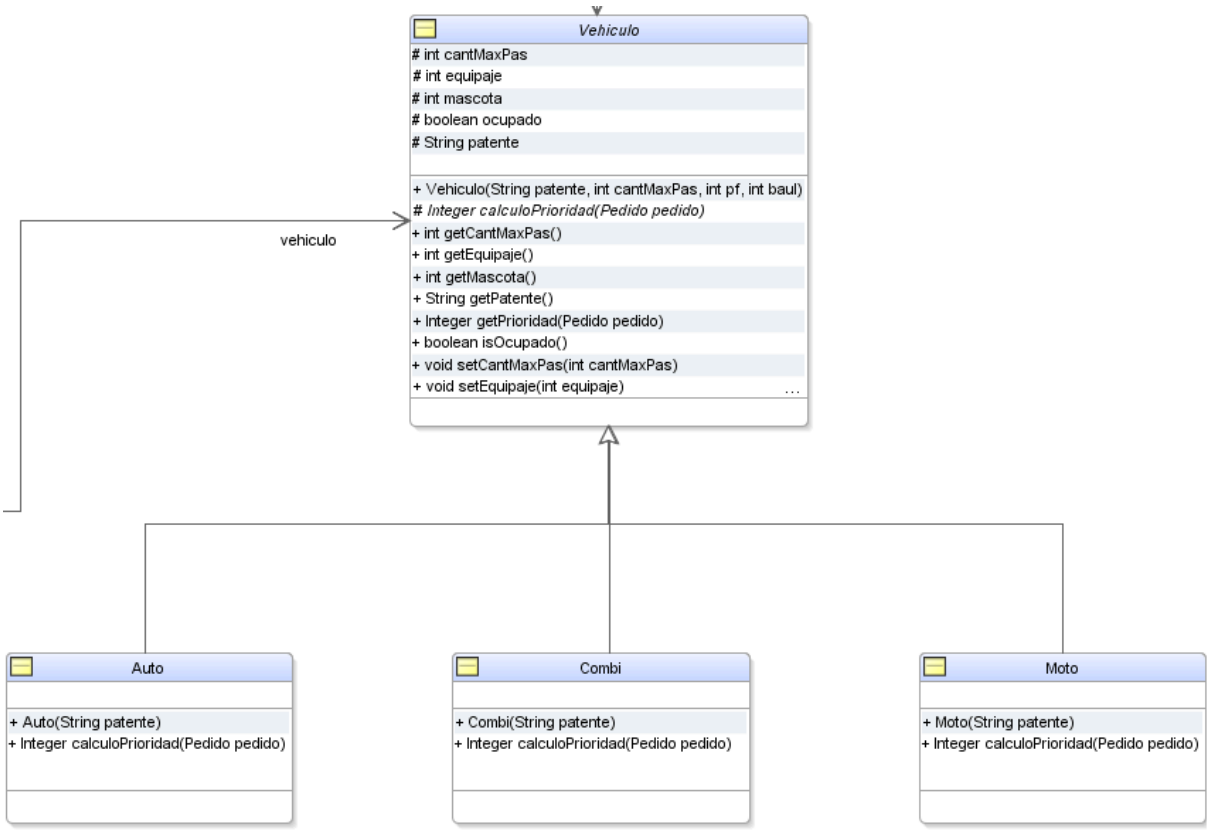
Patron Template

El patrón template se basa principalmente en el hecho de que si un algoritmo puede aplicarse a varios supuestos en los que únicamente cambie un pequeño número de operaciones, la idea será utilizar una clase para modelarlo a través de sus operaciones. Esta clase base se encargará de definir los pasos comunes del algoritmo, mientras que las clases que hereden de ella implementarán los detalles propios de cada caso concreto, es decir, el código específico para cada caso.

Para este trabajo, la inclusión de este patrón facilitó un aspecto fundamental relacionado a los vehículos de la empresa. La empresa no asigna cualquier vehiculos a cualquier viaje, sino que adopta un criterio en base a los requerimientos del viaje (cantidad de pasajeros, mascota, equipaje) y evalúa cada uno de los vehículos en base a estos requerimientos para ver cuale s el que mejor se ajusta a las necesidades particulares de un cliente.

Por ello, fue necesario para cada vehículo, determinar una serie de operaciones que, en base a los atributos de viaje recién mencionados, devolviera un valor representativo que informara justamente lo bien que se ajusta el vehículo a tal viaje.





Para el tratamiento de los vehículos de nuestra empresa, definimos un ancestro abstracto común (***Vehiculo***) y aplicamos uno de los patrones creacionales más útiles en el panorama de la programación orientada a objetos: el patrón Factory. Este resulta muy importante, ya que además de centralizar la lógica de creación de los vehículos en un solo lugar y evitar que estos se creen en cualquier ”zona” del código, nos permite tener una flexibilidad al momento de, en un futuro, agregar la creación de nuevos tipos de vehículos. Esto lo podríamos hacer modificando la misma clase o, también, creando una nueva clase que herede de esta y escribiendo el código en la misma.

En lo que respecta a la clase ***Vehiculo*** ,la misma es una clase abstracta de la cual heredan los diferentes tipos de vehículos (las clases concretas ***Moto***, ***Auto*** y ***Combi***) sus atributos, y la cual permite a las mismas implementar los métodos que sean específicos de cada uno.

Una de las partes más interesantes de estas clases es el método *getPrioridad****()***. El mismo devuelve un ***Integer*** que, dado un pedido, indica qué tan apto es el vehículo para llevarlo a cabo y, en el caso de que no lo sea, devuelve null. En este método implementado en la clase ***Vehiculo*** aplicamos el patrón Template para definir la estructura básica del cálculo de prioridad. Esto nos permitió dejar que las clases hijas puedan definir de formas específicas según sus características los métodos necesarios para su cálculo, y así evitar la sobreescritura de la operación entera. Este nos permite abstraer el proceso común de evaluación de prioridad y, al mismo tiempo, nos brinda la flexibilidad de extender o modificar este proceso en las clases hijas del mismo.

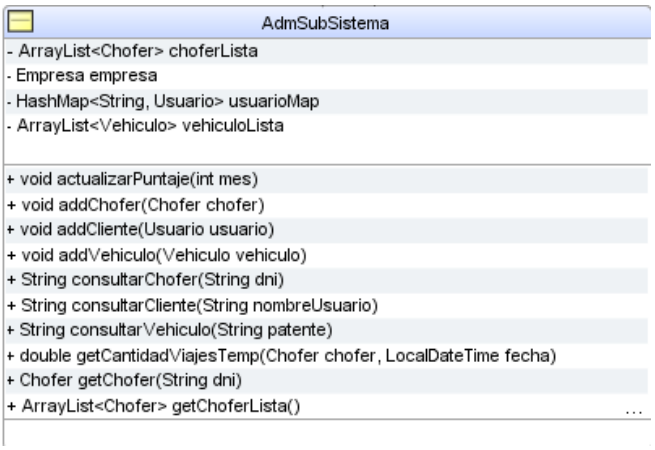
Patrón Singleton y Facade

En la primer parte, la clase encargada de ejecutar la lógica del negocio correspondiente a la capa de negocio de la arquitectura de 3 capas, fue nuestra clase ***Empresa***, la cual a su vez contaba con dos clases ***AdmSubSistema*** y ***viajesSusbSistema*** para poder llevar adelante las solicitudes de los distintos actores del mundo del problema.

En esta segunda parte, si bien no se comunican los hilos directamente con la empresa para hacer sus solicitudes, la clase recurso compartido delega la realización de tales solicitudes a ella. Esto permite aprovechar los métodos desarrollados durante la primer parte, así como permitir una visión más clara y limpia de las zona crítica del recurso compartido, justamente desarrollando la clase ***RecursoCompartido,*** la cual realmente implementa métodos synchronized

Respecto a la clase ***Empresa***, la implementamos aplicando el patrón Singleton, pues es necesario una única instancia de nuestra empresa.También se hizo uso del patrón Facade. Este último nos permite dirigir las peticiones a la ***Empresa*** y que ésta actúe de fachada, es decir, se encargue de recibir las solicitudes y de ocultar cómo se resuelven. De esta forma, cuando se quiera solicitar una determinada funcionalidad, como pedir un viaje, bastará con enviar el mensaje a la ***Empresa,*** a través de la invocación del método correspondiente, y ella se encargará de invocar los métodos que sean necesarios para la generación de un pedido y un posible viaje posterior.

AdmSubSistema



Para llevar adelante las funcionalidades, la ***Empresa*** cuenta con dos clases las cuales mantienen una relación de composición con la misma. La primera de ellas es la clase ***AdmSubSistema***.

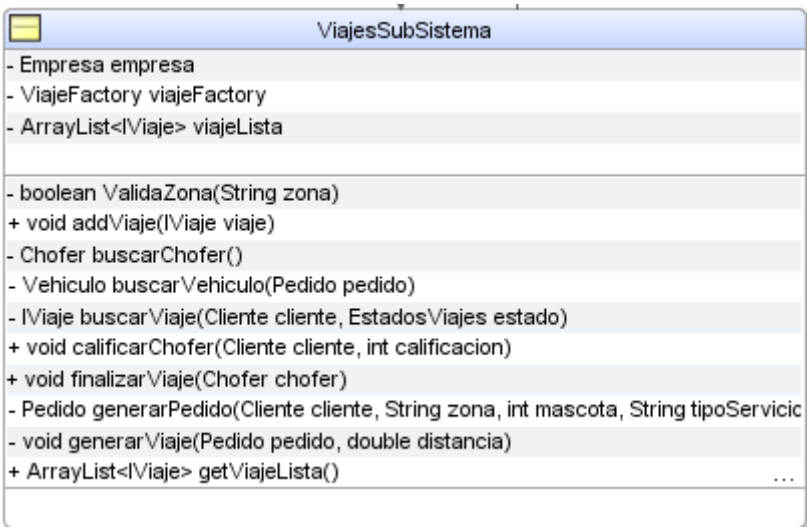
Esta clase se encarga de las funcionalidades que tiene el administrador. Entre ellas encontramos:

- ➔ Añadidura, consulta y modificación de usuarios, choferes y vehículos
- ➔ Solicitud de listado de usuarios, choferes, vehículos y viajes (ordenados por costo de mayor a menor)
- ➔ Actualización de puntaje de los choferes de acuerdo a los viajes realizados y los Km recorridos
- ➔ Generación de reportes de clientes y choferes con detalle de sus sueldos
- ➔ Generación de reportes de viajes realizados por un cliente o chofer entre dos fechas distintas
- ➔ Cálculo del total de dinero necesario para pagar a los choferes

Por supuesto, la clase ***AdmSubSistema*** posee las colecciones de usuarios, choferes y vehículos que pertenecen a la empresa de transporte. De esta manera, cuenta con los métodos necesarios para llevar adelante las tareas mencionadas, y se vale de estas colecciones para iterar sobre ellas, añadir objetos o modificar alguno de los que ya posean.

A su vez, los parámetros que utilizan cada uno de estos métodos son previamente validados en la clase ***Empresa***, la cual recibe una solicitud, revisa que los datos sean adecuados y llama los métodos necesarios para cumplirla.

ViajesSubSistema



La otra clase con la que cuenta la ***Empresa*** para llevar adelante las solicitudes que reciba, es la clase ***ViajesSubSistema***. Como bien lo indica su nombre, la función básica de esta clase es encargarse del tratamiento de los viajes de nuestra empresa y, por ende, de las funcionalidades de los clientes. Entre ellas encontramos:

- ➔ Completar un formulario de solicitud de viaje
- ➔ Pagar un viaje
- ➔ Calificar un chofer
- ➔ Visualizar sus viajes

Además, la clase ***ViajesSubSistema*** es la que posee la colección de viajes de la empresa de transporte.

PERSISTENCIA

La persistencia en el proyecto se basa en la serialización de objetos Java para almacenar y recuperar datos de forma persistente. Se utilizaron dos mecanismos de serialización:

- **Serialización binaria:** Implementada por la clase *PersistenciaBIN*, guarda objetos en formato binario en archivos.
- **Serialización XML:** Implementada por la clase *PersistenciaXML*, guarda objetos en formato XML en archivos.

La elección del mecanismo de serialización depende de las preferencias del desarrollador, aunque nosotros optamos por la persistencia en XML por su facilidad para alterar o actualizar los datos ya guardados.

Patrón DTO / DAO

Clases DTO

Las clases DTO (Data Transfer Object) se utilizan para transferir datos entre la capa de negocio y la capa de persistencia. Estas clases encapsulan los atributos relevantes de los objetos de dominio/modelo y los convierten en un formato adecuado para la serialización. Se debe destacar la importancia en que todas las clases DTO implementan Serializable para ser posible la serialización a través de los métodos planteados en IPersistencia.

Se desarrollaron las siguientes clases DTO:

- ***ChoferDTO*:**
  - *ChoferContratadoDTO.*
  - *AsalariadoDTO*.
  - *ChoferPermanenteDTO*.
  - *ChoferTemporarioDTO*.
- **VehiculoDTO:**
  - *AutoDTO.*
  - *MotoDTO.*
  - *CombiDTO.*
- **UsuarioDTO:**
  - *ClienteDTO.*
  - *AdministradorDTO*.
- **EmpresaDTO.**
- **AdmSubSistema.**
- **ViajesSubSistema.**
- **ViajeDTO.**
- **PedidoDTO.**
- **LocalDateTimeDTO.**

También desarrollamos una clase específica LocalDateTimeDTO para la persistencia de las fechas debido a que la librería ***LocalDateTime*** carece de un constructor vacío nos era imposible serializarla, por lo que recurrimos a una clase propia con el constructor, getters y setters necesarios para persistir y transferir los datos de fecha según la necesidad.

Clases DAO

El proyecto implementa un patrón DAO (Data Access Object) de forma que si bien no desarrollamos clases DAO específicas para cada entidad, hemos implementado la interfaz *IPersistencia* la cual define métodos para serializar y deserializar objetos, mientras que las clases concretas ***PersistenciaBIN*** y ***PersistenciaXML*** son las responsables de implementar estos métodos para cada tipo de serialización: *abrirInput(), abrirOutput(), cerrarOutput(), cerrarInput(), escribir() y leer().*

Por su parte las clases ***UTIL*** no encapsulan directamente el acceso a la capa de persistencia. Se basan en dicha serialización para guardar y cargar datos. Su función principal es la conversión entre objetos de dominio y DTOs, lo que las convierte en clases de mapeo, representan el nexo entre la capa de lógica de Negocio y la capa de persistencia:

- ***UTILEmpresa***.
- ***UTILAdmSubSistema***.
- ***UTILViajesSubSistema***.

En el caso de ***UTILAdmSubSistema*** abarca los métodos para la conversión de choferes, vehículos y usuarios. Así como de la propia clase *AdmSubSistema*, siendo responsable de crear un ***AdmSubSistemaDTO*** a partir del Object *AdmSubSistema*, replicando y convirtiendo a su vez todas las colecciones y objetos que contiene la clase para poder persistirla.

Lo mismo ocurre en ***UTILViajesSubSistema*** pero abarcando pedidos, viajes y la propia clase ViajesSubSistema junto con su colección de viajes históricos.

En el caso de ***UTILEmpresa*** además tuvimos que considerar un constructor público debido que al implementar la clase Empresa el Patron singleton, entonces carecía de uno desde el principio. Esta clase al carecer de muchos atributos se centra en la conversión a DTO y viceversa de los subsistemas de Adm y Viajes, como así del Administrador que finalmente ha caído en desuso.

MODIFICACIONES

Respecto a la entrega de la primera parte del trabajo, fueron necesarias algunas modificaciones en pos de obtener un mejor funcionamiento durante esta segunda entrega y corregir errores de la primera.

- En primer lugar, implementamos una clonación profunda de la lista de viajes, junto con todos sus atributos, de forma tal que la lista clonada quede libre de ser modificada a gusto sin corromper la lista original

- Eliminamos modificador static de atributos de la empresa, (***AdmSubSistema*** y ***viajeSubSistema***) los cuales verdaderamente no estaban planteados para usarse de esa manera (ni se usaron), pero por algún descuido habían quedado declarados de ese modo

- Solucionamos errores de uso de la ***Empresa*** como variable global en el cálculo de los sueldos de los choferes. Antes, cada chofer en su método *getSueldo()* invoca a la empresa y solicitaba lo que necesitaba para realizar ese cálculo (su cantidad de viajes en el mes por ejemplo). Sin embargo, en esta segunda etapa, con el fin de evitar este mal uso de la variable, se reescribieron los métodos de forma tal que cada chofer reciba por parámetro los atributos que requiera para calcular su sueldo, evitando de esa forma invocaciones erróneas de la clase Empresa.

CONCLUSIÓN

En esta segunda etapa del trabajo práctico nos enfocamos en la implementación de concurrencia y en el desarrollo de una interfaz de usuario realista. Este enfoque nos permitió simular comportamientos más realistas en cuanto al acceso a recursos y la interacción del usuario.

Optamos por una metodología de trabajo grupal en la que cada decisión de diseño fue consensuada, lo que permitió abordar cada aspecto del proyecto de manera integral y colaborativa. Esta estrategia evitó conflictos comunes y permitió identificar y corregir errores de forma temprana, asegurando un avance más firme y efectivo incluso habiendo perdido un miembro del equipo.

Implementamos una arquitectura modelo-vista-controlador (MVC), adaptando la estructura inicial de tres capas. La concurrencia fue manejada a través de hilos de 3 clases que extienden Thread y una clase RecursoCompartido que actúa como monitor, regulando el acceso a los recursos para evitar inconsistencias y pérdidas de datos. El patrón Observer facilitó la comunicación de cambios en el recurso compartido a las vistas, simplificando la actualización de múltiples ventanas.

El uso del patrón Decorator permitió una extensión flexible de la funcionalidad, especialmente en el cálculo del costo de los viajes, evitando métodos sobrecargados y mejorando la claridad del código. El patrón Template nos ayudó a definir un algoritmo base para la asignación de vehículos, adaptándose a los requerimientos específicos de cada viaje.

Además, aplicamos los patrones Singleton y Facade en la clase Empresa, asegurando una única instancia y simplificando la interacción con la lógica del negocio. La persistencia se logró mediante serialización binaria y XML, facilitando el almacenamiento y recuperación de datos de forma eficiente.

En resumen, la combinación de una metodología colaborativa, el uso de patrones de diseño adecuados y una implementación cuidadosa de la concurrencia y la persistencia nos permitió desarrollar una solución robusta y eficiente para el proyecto.