UNMdP – F.I. – Ingeniería en Informática

Programación III

“Subí que te llevo”

Grupo N°8

Bautista Rodriguez
Agustin Chazarreta
Thiago Parise

Año 2024

ÍNDICE

RESUMEN

Solicitud

Se requiere el desarrollo de un sistema informático para una empresa de traslados de pasajeros y de mensajería. La interfaz de usuario será una computadora, los requerimientos serán limitados.

Descripción breve del sistema

El sistema debe gestionar parte de la información de una empresa de transporte de pasajeros y mensajería.
La Empresa cuenta con una flota de Vehículos de diferentes características, un conjunto de Choferes que manejan cualquiera de los Vehículos y un conjunto de Clientes registrados con los cuales opera. Estos actores y recursos son dinámicos en el tiempo, o sea, pueden aumentar.
A modo de ejemplo se presentan estos casos:
Casos de uso :

1. Un Cliente registrado completa el formulario de solicitud de Viaje (Pedido) En ese momento se crea un Viaje en estado “solicitado”.
2. El sistema deberá permitir la creación de un viaje (Pedido) y mostrar su evolución en el tiempo siguiendo la cronología propuesta en la sección “Evolución del Viaje desde que nace como Pedido”

1. INTRODUCCIÓN

1.1. OBJETIVOS DEL TRABAJO

El objetivo del trabajo es simular el funcionamiento de una empresa de viajes mediante la aplicación de patrones de diseño, la concurrencia y una interacción con una interfaz gráfica. A partir de una configuración, la simulación genera una determinada cantidad de choferes, clientes y vehículos; los cuales funcionarán de manera automática y concurrente, existiendo también la posibilidad de interactuar con una interfaz gráfica para simular un cliente que se registra en la empresa o inicia sesión para realizar pedidos específicos.

1.2. MOTIVACIONES

Mediante la realización de este trabajo, pretendemos profundizar los contenidos vistos durante las clases, respecto a la aplicación de los patrones de diseño, la programación concurrente, el uso de interfaces gráficas, el manejo de excepciones, el uso de contratos y demás contenidos. Con este trabajo, también pretendemos reforzar el trabajo en equipo, la comunicación entre los miembros del mismo, la distribución de tareas, el análisis en conjunto del problema y el diseño en conjunto de una solución para el mismo.

2. DESARROLLO

2.1. METODOLOGÍA

2.2.I. PASOS

En un principio lo que hicimos fue definir una estructura del proyecto, planteando responsabilidades de clases, comportamientos y características en forma general permitiéndonos una mejor organización y orientación en el desarrollo del trabajo.

Luego seguimos avanzando tanto con implementaciones individuales y reuniones grupales mediante la aplicación Discord en donde volcamos nuestros avances y conversábamos sobre lo que íbamos aplicando.

Al mismo tiempo llevamos el desarrollo del trabajo junto con un documento en común en el que escribíamos las dudas para luego consultarlas con los profesores.

2.2. DISEÑO E IMPLEMENTACIÓN

2.2.I. DIAGRAMA UML

Una vez terminado el trabajo, utilizamos JDeveloper para generar el diagrama UML con todas las clases involucradas en el trabajo.
Link para descargar la imagen del UML: https://postimg.cc/R6Gr0qkW

2.2.II. DISEÑO MVC

Para la segunda parte del trabajo se aplicó el patrón MVC (Model View Controler), el cual consta de 3 elementos:

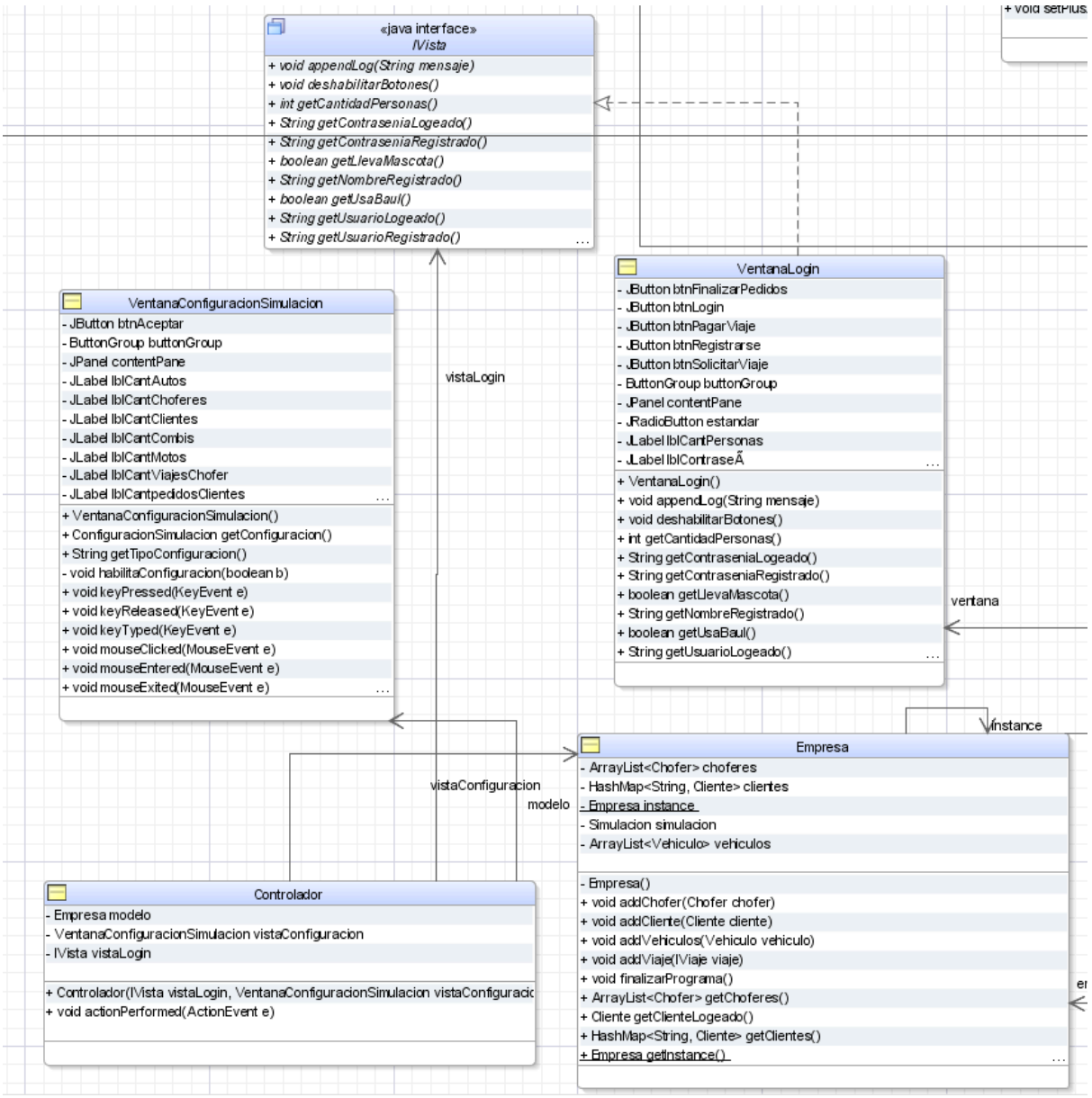
VISTA: Para este caso, existen 3 ventanas: una para configurar la simulación (VentanaConfiguracionSimulacion), otra para visualizar el avance de los viajes (VentanaGeneral) y otra que permite registrar/loguear a un usuario y hacer pedidos específicos (VentanaLogin). La ventana de seguimiento de los viajes (VentanaGeneral) no lleva un controlador ya que no interactúa con el usuario y tampoco genera eventos que necesiten ser "escuchados". Las otras dos ventanas sí cuentan con un controlador que gestiona los eventos realizados en las mismas.

CONTROLADOR : Para este caso existe un controlador que gestiona los eventos de dos ventanas, la ventana que permite registrar/loguear y hacer pedidos específicos; y la ventana de configuración de simulación. Funciona como intermediario entre dichas ventanas y el modelo. Desde este controlador, se toman los datos de configuración de simulación de la respectiva ventana y se envían al modelo para configurar la simulación; y también se toman los datos del pedido específico del usuario logueado y se envían al modelo para que él mismo lo gestione.

MODELO: El modelo está compuesto por todas las clases que representan la lógica de la solución. En el modelo se encuentra la clase Empresa que contiene a las distintas entidades del programa, como el objeto simulación que se encarga

de manejar lo relacionado al recurso compartido (los métodos a los que acceden los hilos de clientes y choferes), los listados de choferes, clientes, vehículos y viajes de la empresa y los métodos necesarios para interactuar entre si.

En la primera parte se trabajó con una arquitectura de 3 capas: Presentación, Lógica de Negocio y Datos. La capa de presentación era la propia clase main, donde se instancian todos los elementos de la simulación. Todas las funcionalidades estaban en la lógica de negocio y los datos los guardaba la empresa.



Con la incorporación del patrón MVC se reemplazó la capa de presentación con la Vista, y la lógica de negocio y datos con el Modelo; y el controlador funciona como intermediario entre ambos.

2.2.III. PATRONES UTILIZADOS:

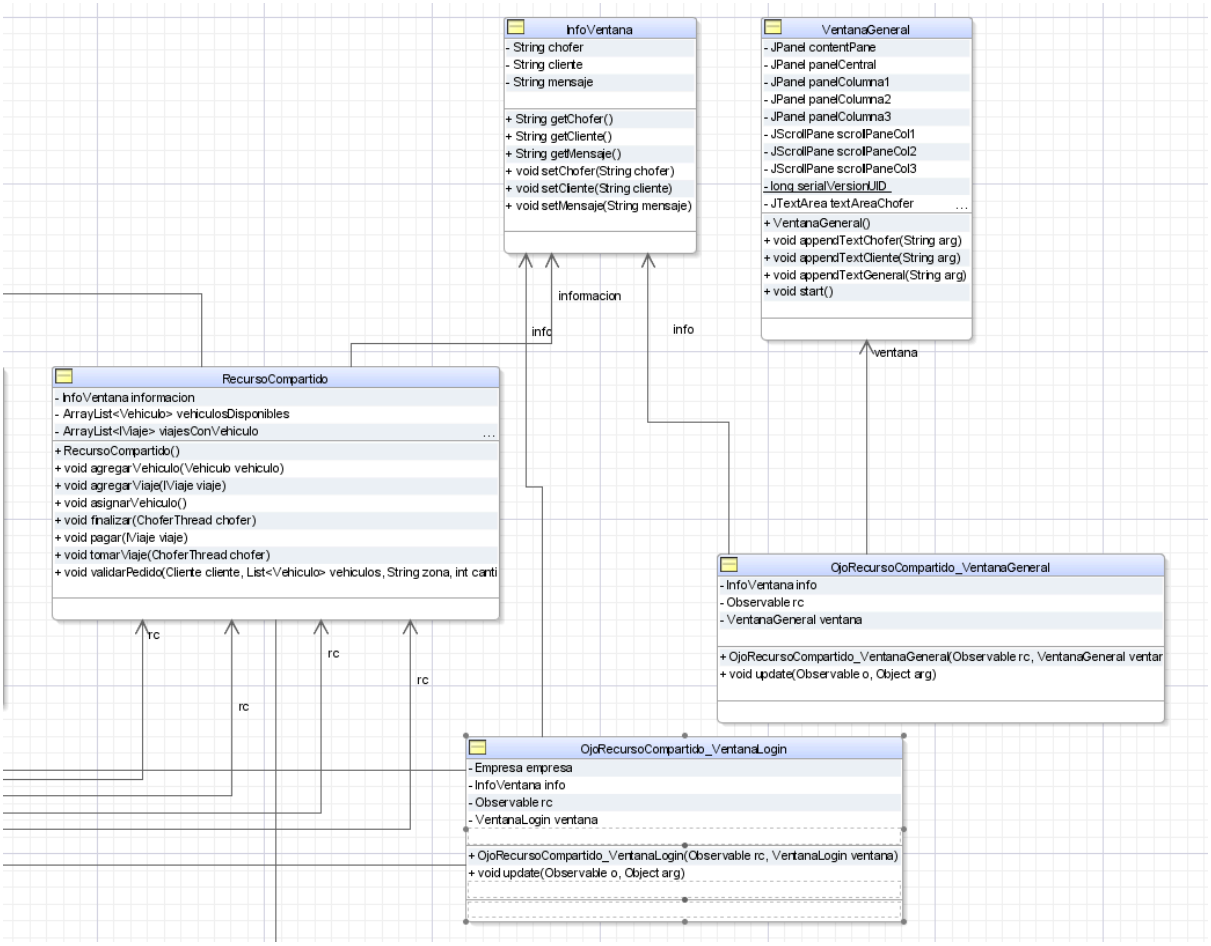
- OBSERVER – OBSERVABLE

El patrón Observer – Observable se utilizó para mostrar los cambios ocurridos en el recurso compartido en las respectivas ventanas. El recurso compartido funciona como observable, y es observado por dos observers. Cada vez que un cliente o chofer hacen uso del recurso compartido, el mismo notifica a sus observers sobre dicho cambio y envía información sobre el cambio. Esto lo hacen las sentencias setChanged() y notifyObservers(), en ese orden. Los cambios que se podrían producir en el recurso compartido se dan por el acceso de los hilos al mismo. Cuando un hilo bloquea el recurso compartido y genera un cambio (asigna un vehículo a un viaje, toma un viaje, paga un viaje o finaliza un viaje), dicho cambio es notificado a los observers por el recurso compartido.

Los observers reciben las notificaciones provenientes de su observable, y muestran dicha información en la ventana correspondiente. Cada Observer observa, valga la redundancia, al recurso compartido y muestra mensajes a una ventana determinada. Existe un observer que muestra mensajes en la "ventanaGeneral", que solo muestra el seguimiento de todos los viajes y los correspondientes a un chofer y cliente predefinido; y un observer que muestra mensajes en la "ventanaLogin", donde se visualiza el avance del viaje del usuario que interactúa con dicha ventana.

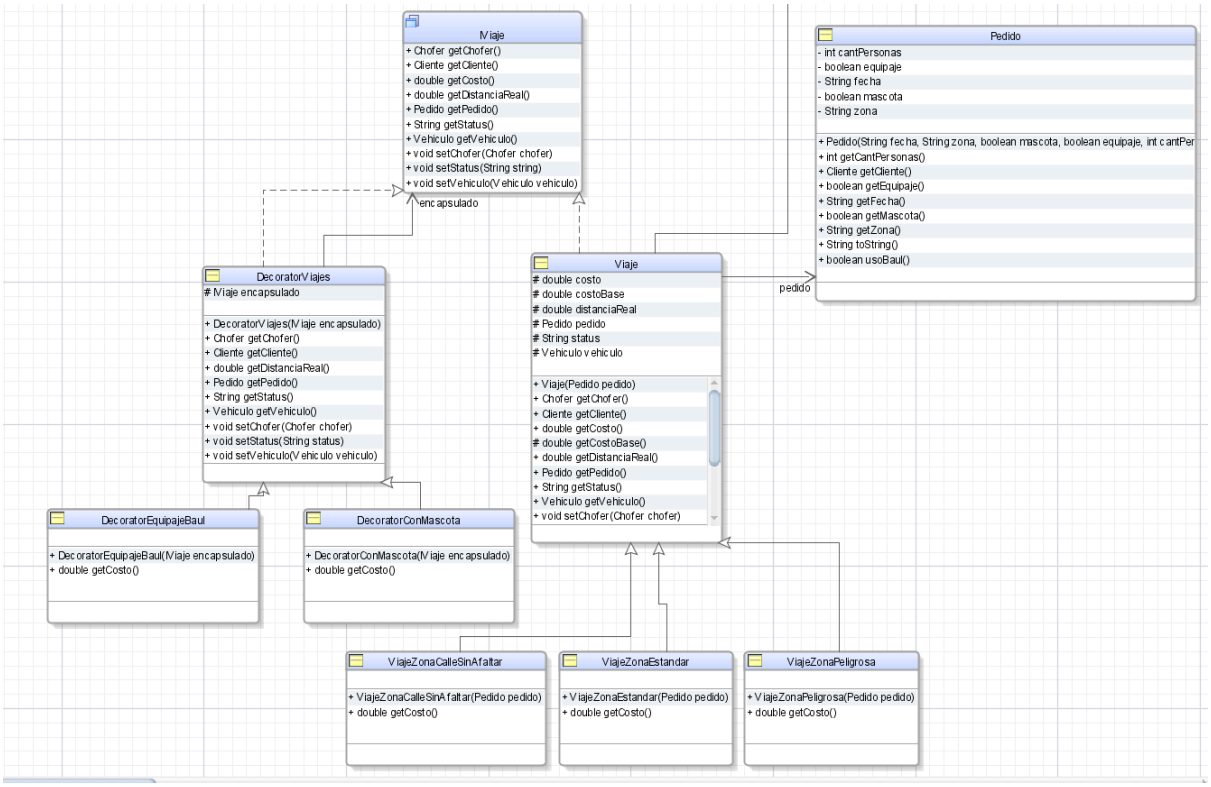
Mediante el método update(), presente en cada observer, se filtra la información y se muestra no solo en la ventana que corresponda, sino que se muestra en las partes de las ventanas reservada para dicha información.

Cada observer se vincula con su observable mediante la sentencia addObserver(), que se ejecuta en el constructor de cada uno. De esta manera el update() de cada observer se ejecutará cada vez que el observable ejecute setChanged() seguido de notifyObservers().



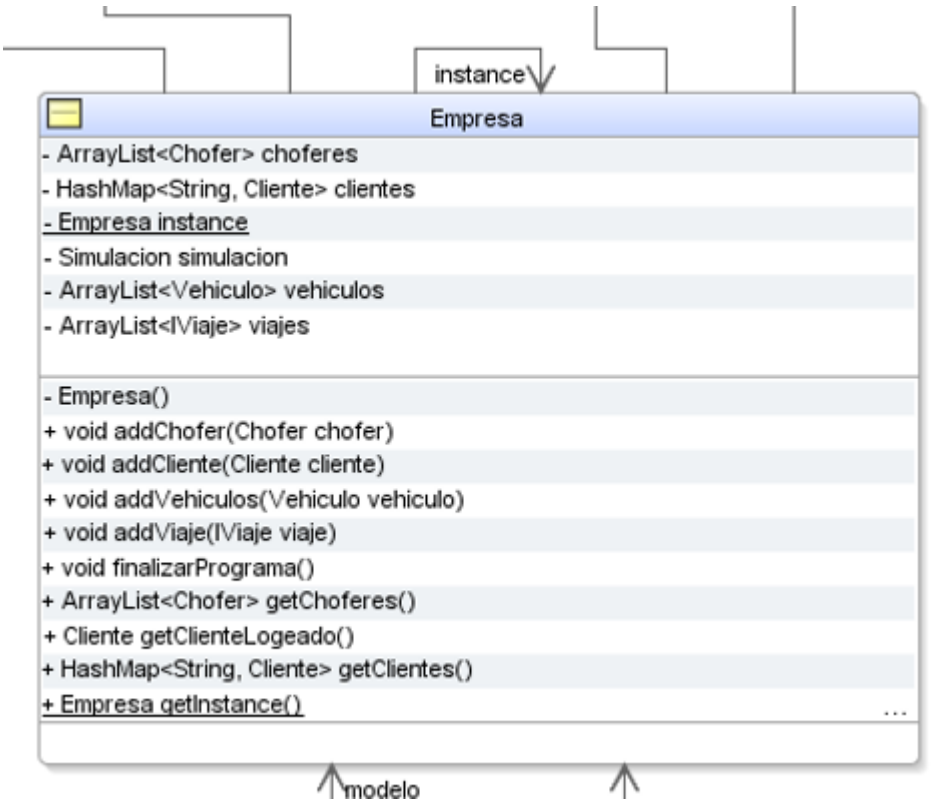
- DECORATOR

El patrón decorator se implementó mediante una clase abstracta “DecoratorViajes”, la cual modela el comportamiento común de los encapsulados y dos clases concretas “DecoratorConMacota” y “DecoratorEquipajeBaul”; las cuales decoran el viaje. Decidimos usar solo dos decoradores ya que esos dos son los únicos que agregan valor al viaje. Cada decorator tiene un atributo de tipo IViaje llamado “encapsulado”, el cual utilizan para obtener el costo que tienen que decorar según sus características. Para esto implementan, al igual que la clase abstracta “Viaje”, la interfaz “IViaje” que les da acceso al método “getCosto()”

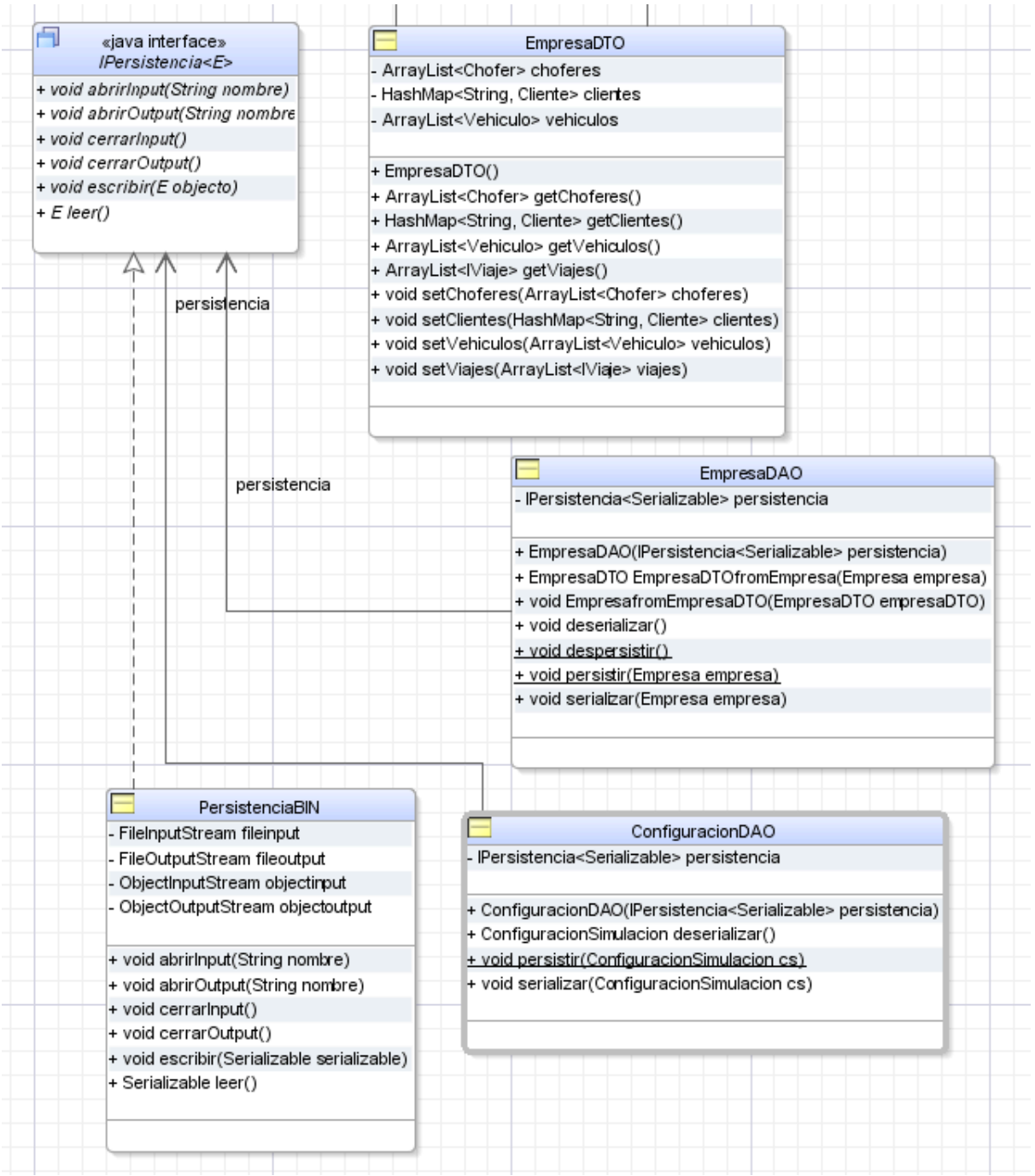


- FACADE / SINGLETON

  El singleton se aplicó a la clase empresa. En nuestro diseño no hay necesidad de tener más de una instancia de la empresa por lo que se aplicó dicho patrón para dicha clase. Para garantizar una única instancia de la empresa, se usó un constructor privado y un método static que devuelve una instancia de la empresa si la misma no existe, o la instancia existente. Además, esta clase aplica el patrón FACADE, que implementa los métodos necesarios para quienes lo necesiten y delega la implementación a la clase que se especializa en resolver esa tarea.



- DAO/DTO
  Mediante un objeto DAO para la empresa y otro para la configuración de la simulación, implementamos métodos para persistir y leer datos persistidos mediante objetos DTO. De esta manera, la empresa no tiene la necesidad de crear objetos nuevos cada vez que inicia el programa, sino que los toma de el archivo que se persiste la última vez que el programa se ejecutó. Acá consideramos que si en la configuración de la simulación se utilizan menos entidades que en la configuración de la última ejecución del programa, esta simulación va a tomar de la empresa las entidades que necesite y, si en cambio se desean usar más entidades que la última ejecución, entonces tomará las entidades necesarias de la empresa y se instanciarán nuevas entidades para cumplir con lo indicado en la configuración.


«java interface»
IPersistencia<E>
+ void abrirInput(String nombre)
+ void abrirOutput(String nombre
+ void cerrarInput()
+ void cerrarOutput()
+ void escribir(E objecto)
+ E leer()
EmpresaDTO
- ArrayList<Chofer> choferes
- HashMap<String, Cliente> clientes
- ArrayList<Vehiculo> vehiculos
+ EmpresaDTO()
+ ArrayList<Chofer> getChoferes()
+ HashMap<String, Cliente> getClientes()
+ ArrayList<Vehiculo> getVehiculos()
+ ArrayList<IViaje> getViajes()
+ void setChoferes(ArrayList<Chofer> choferes)
+ void setClientes(HashMap<String, Cliente> clientes)
+ void setVehiculos(ArrayList<Vehiculo> vehiculos)
+ void setViajes(ArrayList<IViaje> viajes)
persistencia
persistencia
EmpresaDAO
- IPersistencia<Serializable> persistencia
+ EmpresaDAO(IPersistencia<Serializable> persistencia)
+ EmpresaDTO EmpresaDTOfromEmpresa(Empresa empresa)
+ void EmpresafromEmpresaDTO(EmpresaDTO empresaDTO)
+ void deserializar()
+ void despersistir()
+ void persistir(Empresa empresa)
+ void serializar(Empresa empresa)
PersistenciaBIN
- FileInputStream fileinput
- FileOutputStream fileoutput
- ObjectInputStream objectinput
- ObjectOutputStream objectoutput
+ void abrirInput(String nombre)
+ void abrirOutput(String nombre)
+ void cerrarInput()
+ void cerrarOutput()
+ void escribir(Serializable serializable)
+ Serializable leer()
ConfiguracionDAO
- IPersistencia<Serializable> persistencia
+ ConfiguracionDAO(IPersistencia<Serializable> persistencia)
+ ConfiguracionSimulacion deserializar()
+ void persistir(ConfiguracionSimulacion cs)
+ void serializar(ConfiguracionSimulacion cs)

2.2.IV. DISEÑO DE CLASES

- PATRÓN DECORATOR: Para este patrón decidimos implementar solo dos decorator, uno correspondiente al uso de baúl y otro correspondiente al transporte de mascota; ya que solo esas dos características del viaje agregan valor al costo inicial. El costo inicial, en nuestro diseño, depende de la zona elegida para el viaje, y sobre dicho costo se agregan los decorados. En un principio implementamos 4 decoradores: los ya mencionados, otro para los viajes con equipaje manual y otro para viajes sin transporte de mascota; pero como ya se mencionó, estas últimas dos características no modificaban el costo del viaje así que decidimos prescindir de las mismas.

- USO DE VENTANAS: Si bien el enunciado pedía 3 ventanas para hacer seguimiento a los viajes, una ventana general para el seguimiento de todos los viajes, otra para el seguimiento de los viajes de un chofer preconfigurado, y otra para el seguimiento de los viajes de un cliente preconfigurado; decidimos, para mejor organización, unificar las 3 ventanas en una sola con 3 áreas de texto, un área para cada una de las funcionalidades mencionadas, a fin de tener una ventana de seguimiento de viajes y una ventana de interacción con el usuario.

- BOTÓN LOGIN Y BOTÓN REGISTRARSE: Cuando un cliente completa los campos de registro y presiona el botón “Registrarse”, si el nombre de usuario no existen se da de alta al nuevo usuario y seguidamente se lo contabiliza como un cliente activo, habilitando la ventana correspondiente para que el mismo haga pedidos. O sea que no es necesario que un usuario recién registrado “inicie sesión” poniendo su usuario y contraseña, y presionando el botón “Login”; esto último se hace automáticamente luego de presionar “Registrarse”.

- CLASE SIMULACIÓN: Es la clase encargada de modelar la simulación intentando simular el funcionamiento del sistema creando robots (threads) que interactúan concurrentemente. La clase cuenta con una referencia al recurso compartido, a la configuración de la simulación y al usuario

logueado en el sistema y su respectivo viaje actual para lograr la concurrencia con los threads robots y el thread cliente.

2.2.V MÉTODOS RELEVANTES

- public void iniciarSimulacion(Empresa empresa)

- Este método carga la simulación con los datos necesarios para su funcionamiento (vehículos, choferesThread, clientesThread).
- Precondiciones: Empresa diferente de null. Al momento de ejecutar este método, la empresa ya debe estar instanciada.

- public void actionPerformed(ActionEvent e)

- Este método gestiona los eventos de las ventanas que tiene como referencia, vinculando dichos eventos con las funcionalidades del modelo.

- public synchronized void asignarVehiculo()
- public synchrinized void tomarViaje(ChoferThread chofer)
  - Precondición: parámetro chofer diferente de null, ya que este método es ejecutado por un choferThread.
- public synchronized void pagar(IViaje viajr);
  - Precondición: parametro viaje diferente de null.
- public synchrinized void finalizar(ChoferThread chofer)
  - Precondición: parametro chofer diferente de null.
- public synchronized void agregarViaje(IViaje)
  - Precondición: parámetro viaje diferente de null
  - Excepción: SinChoferesException, lanzada cuando no hay más choferes activos y se intenta agregar otro viaje solicitado.

Estos métodos son relevantes porque representan el funcionamiento concurrente de los threads que intervienen en la simulación.

3. CONCLUSIONES

3.1. Cumplimiento de expectativas

Al comenzar este trabajo, nuestras expectativas estaban orientadas a profundizar los contenidos vistos durante las clases, respecto a la aplicación de los patrones de diseño, la programación concurrente, el uso de interfaces gráficas, el manejo de excepciones, el uso de contratos y demás contenidos. Con este trabajo, también pretendemos reforzar el trabajo en equipo, la comunicación entre los miembros del mismo, la distribución de tareas, el análisis en conjunto del problema y el diseño en conjunto de una solución para el mismo.

En cuanto a los aspectos académicos logramos aplicar y consolidar los temas teóricos vistos en clase.

El uso de los patrones de diseño en el trabajo práctico nos permitió conocer la aplicación de estos dependiendo el contexto y expandir nuestras habilidades a la hora de resolver problemas que ocurren en el diseño de software. La programación concurrente también fue un punto clave, ya que el uso de una simulación nos presentó la difícil tarea de manejar varios hilos de ejecución de manera efectiva. Además el desarrollo de interfaces gráficas nos permitió mejorar nuestra capacidad de crear ventanas más interactivas y completas, acompañado de las dificultades que conlleva utilizar el patrón MVC correctamente.

Más allá de cumplir con nuestras expectativas iniciales, esta experiencia nos brindó mucha más confianza y experiencia para el desarrollo de programas de mayor complejidad.

3.2. Dificultades encontradas y sus soluciones

Durante el desarrollo del trabajo, nos encontramos con algunos problemas a la hora de decidir qué camino seguir en cuanto al diseño y la distribución de responsabilidades entre las clases del programa. Para solucionarlo, utilizando un borrador escribimos las clases que se involucraron y analizamos alternativas entre los participantes,

En cuanto al trabajo colaborativo, en la primera parte tuvimos problemas para lograr una comunicación eficaz entre todos los miembros del grupo lo que perjudicó la realización de la primera entrega. En esta segunda parte mejoramos muchos aspectos que abarcan el trabajo grupal, logramos una muy buena comunicación tanto a la hora de trabajar en la solución de problemas como de comunicar cambios individuales, la comunicación fue imprescindible a lo largo de todo el desarrollo.

3.3. Solución destacada

Para la persistencia de los datos de la empresa al cerrar el programa, creamos un botón oculto en la vista de el cliente humano que, al cerrar la ventana, se simula un click, enviando un evento al controlador para que ejecute el método del FACADE que se encargue de persistir los datos (delegando a los objetos DAO esta funcionalidad), y luego de persistir los datos, llamar a System.exit(0) para finalizar con el programa. Tuvimos que redefinir el default close operation de el JFrame de la ventana de cliente humano para que no haga nada al cerrarse y le agregamos un windowlistener para que, al querer cerrar la ventana, lance el evento al controlador.

3.4. Aprendizaje

Como grupo, creemos que si tuviésemos que volver a realizar esta experiencia, nuestra prioridad sería lograr una mayor comunicación y organización grupal. A lo largo del proyecto, nos dimos cuenta de que una coordinación efectiva y una comunicación clara son esenciales para un desarrollo exitoso. La falta de estas puede llevar a malentendidos, y a una menor eficiencia en el desarrollo del proyecto.

En este sentido, hemos aprendido que parte de nuestra labor como futuros ingenieros informáticos no se limita únicamente a resolver problemas técnicos de manera individual. También implica la capacidad de trabajar de manera efectiva en un entorno de equipo.