Subí que te llevo 2° parte

Grupo 05 - Programación Ⅲ - Ingeniería en Informática

Arce Ezequiel
Esposito Cristian
Galarco Barrios Angelina
Nieto Iván Ezequiel

Facultad de Ingeniería - 2024

REPOSITORIO DE GITHUB : https://github.com/CristianEsposit/Grupo_05

RESUMEN

En esta segunda entrega, se desarrolló una simulación para imitar el comportamiento de una empresa de transporte de pasajeros, también se implementó la vista para un usuario, con el fin de que éste haga un pedido y que la aplicación lo procese. En una ventana paralela se visualiza la relación entre el proveedor y el cliente. Para llevar a cabo esta tarea se implementó una serie de patrones que se detallarán más abajo y se añadió la concurrencia en el código para poder simular el comportamiento de los distintos actores en paralelo.

Asimismo, el sistema ahora es persistente.

Patrones Nuevos Utilizados

- Patrón MVC
- Patrón DAO-DTO
- Patrón Observer-Observable

Introducción

El trabajo se realizó con el objetivo de simular correctamente utilizando concurrencia el comportamiento de una empresa de vehículos similar a Uber, donde se encuentra tanto la interacción de los clientes, choferes y la aplicación.

Con el desarrollo de este trabajo se desarrollarían nuestras habilidades para trabajar con concurrencia y distintos patrones de uso tales como Observer-Observable y MVC.

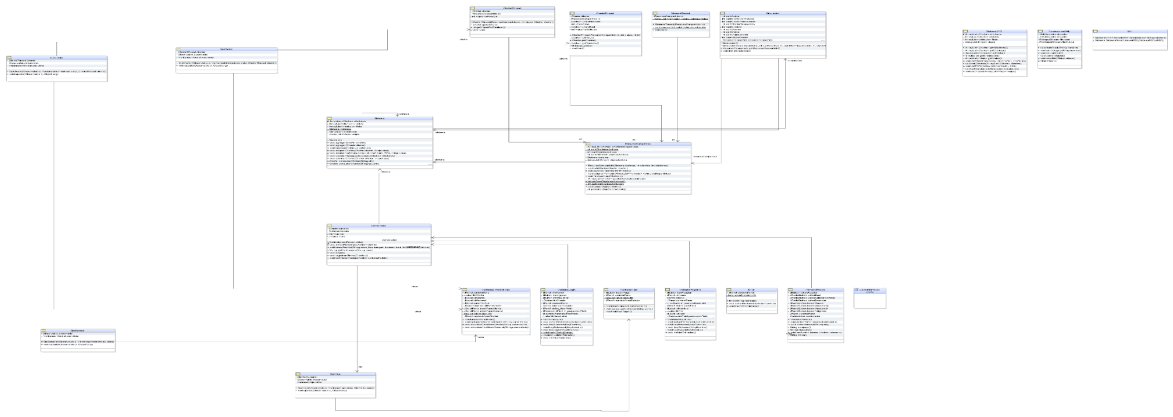
Desarrollo

Metodología

Los principales pasos a seguir para el desarrollo del trabajo:

1. Planteo: Discutir cómo encarar las nuevas funcionalidades y la lógica del viaje, de esta manera logramos que todo el grupo entendiese cómo se implementan las soluciones y pudimos esclarecer dudas que tuviesen lugar acerca de un mecanismo, lógica,etc.
2. División: El siguiente paso fue dividir el trabajo en partes, de esta manera nadie solaparía el trabajo de otro por accidente y se optimizaron los tiempos.
3. Puesta en común: Se unieron las distintas partes del trabajo y logramos que todas funcionen entre sí.

Diseño e implementación



Al utilizar el patrón MVC, se debió dividir el sistema en tres partes principalmente, en modelo, vista y controlador, pero aun en la parte de modelo se mantuvo la división anterior del modelo y negocio, manteniendo en uso el patrón FACADE y la utilización de los patrones de la anterior parte.

También se añadió una sección para la persistencia del programa, en esa parte se implementó el patrón DAO-DTO.

Se debió añadir un paquete de simulación que cuenta con una serie de clases para la creación de los hilos de los clientes, choferes y del sistema, además se añadió un recurso compartido que cuenta con la lista de choferes disponibles y viajes activos. En el propio recurso compartido se encuentra la lógica para administrar el acceso a las listas que se encuentran dentro del mismo.

La parte de persistencia se creó en otro paquete con el mismo nombre, en el cual se crearon las clases para la utilización del patrón DAO-DTO. Decidimos persistir los datos con un formato XML para también poder, a partir del archivo creado, analizar si los datos se persisten de manera correcta.

Simulación

La simulación se inicia mediante la interfaz gráfica (en el log-in).

En primer lugar el programa inicializa aleatoriamente todo lo necesario para su funcionamiento, utilizando como semillas atributos de la clase Simulación.

Entonces se crean un número aleatorio de *ClientesThread* y *ChoferesThread* que interactúan con el sistema y *simulan* la interacción de varios usuarios.

Cada una de las acciones que realizan los distintos hilos “bots” se especifican en una ventana general a medida que se va ejecutando el programa. Incluyendo posibles excepciones.

Patrón MVC

El patrón MVC se utilizó para la división del sistema en tres partes: modelo, vista y control. En la parte de modelo se utilizó como base el negocio y el modelo de la anterior parte del trabajo práctico.

Con respecto a la vista se crearon varias ventanas para la correcta simulación del programa, una ventana informativa que se encuentra dividida en tres secciones; una sección muestra toda la información relacionada con un chofer (hilo) en particular, otra sección muestra toda la información relacionada con un cliente (hilo) en particular y por último una sección general que muestra toda la información relacionada con todos los agentes que forman parte de la simulación. Además hay otras series de ventanas que tienen que ver con el propio usuario como la ventana para registrarse, hacer un login, realizar un pedido y el seguimiento de su viaje.

El controlador al implementar la interfaz ActionListener es el encargado de controlar todas las peticiones del usuario con la ventana, principalmente a través de los botones que están en las mismas.

Patrón DAO-DTO

Este patrón se utilizó para lograr la persistencia de los datos de nuestro sistema. Solo se debió utilizar el DTO para el sistema debido al patrón Singleton que esta clase implementa, para el resto de las clases se añadió los getters y setters públicos de sus atributos y el constructor vacio público, con la finalidad de hacer la persistencia XML de esas clases sin la necesidad de crear más clases del tipo DTO, aunque como consecuencia se tuvo que ensuciar el código al añadir lo anteriormente mencionado.

El SistemaDTO cuenta con todos los requerimientos para utilizar la persistencia de tipo XML. Para pasar de la clase Sistema a la del DTO se utilizó una clase auxiliar llamada Util, la cual su dos únicas funciones es pasar del Sistema a SistemaDTO y viceversa al utilizar los métodos estáticos.

Patrón Observer-Observable

Se implementó este patrón para la actualización de algunas vistas en la ventana. Se crearon las clases llamadas OjoGeneral, OjoCliente y OjoChofer, éstas observan al recurso compartido y contienen una referencia a la vista que actualiza dependiendo de su función, por ejemplo, el OjoCliente sólo actualiza la información de la vista cliente si el cambio que sucedió en el recurso compartido tiene que ver con el mismo cliente que está mostrando la vista del cliente.

Para llegar a esto se debió hacer que las clases de Ojo implementen la interfaz Observer y que el recurso compartido se extienda de la clase Observable.

También se debió crear un cuarto ojo para la actualización de la vista del usuario una vez que se le acepte un pedido y que este se vea actualizado mediante las actualizaciones del recursos compartido.

Diseño de clases

Clases Ojos

Existen 4 clases con el prefijo Ojo en sus nombres, estas se tratan de los observadores. Todas observan al recurso compartido y reciben una notificación cada vez que este se ve modificado, la diferencia entre cada una radica en que guardan una referencia a lo que les interesa saber si se modificó. Mediante el método update en el segundo argumento viene un mensaje, y a partir de este el ojo lo analiza y si contiene algún dato de la referencia que guarda entonces muestra el mensaje en la ventana correspondiente. Por ejemplo, el OjoCliente contiene la referencia a un solo cliente, si el recurso compartido en el notifyObservers manda un mensaje con el nombre de ese cliente, el OjoCliente va a actualizar la vista Cliente, sino lo contiene no va a hacer nada.

Clases Ventanas

Existen varias clases de Ventanas en el paquete vista, cada una cumple con la función de interactuar con el usuario o mostrarle información que pueda interesarle. La ventana llamada VentanaLogin cumple con la función de que el usuario pueda hacer un login o registrarse, incluso permite iniciar la simulación, si el usuario desea registrarse irá a una ventana diferente llamada VentanaRegistro.

También se encuentra la VentanaPedido, en esta el usuario puede realizar el pedido del viaje que desee, este pedido entraría dentro del recurso compartido y sería tratado como uno más.

Las tres ventanas anteriormente mencionadas(VentanaPedido, VentanaRegistro y VentanaLogin) implementan ActionListener, debido a que deben validar los datos que el usuario ingreso, si esto no se realiza esta responsabilidad caería sobre el controlador.

La VentanaInformativa es quien muestra la información tanto de un chofer en particular, un cliente en particular y toda la información de lo que pasa en el sistema, no cumple ninguna otra función más que esa, por lo tanto el controlador desconoce de ella. La forma en que se actualiza es mediante los ojos anteriormente mencionados, ya que estos guardan una referencia de ella, ya que no observa a la vista sino al recurso compartido.

Clase Controlador

La clase controlador es la encargada de controlar las peticiones que hace el usuario, cumple la función de puente entre la vista y el modelo. Además es quien trata la petición de iniciar la simulación, debido a esta función debe implementar ActionListener como las ventanas.

Clases SistemaThread

La clase SistemaThread implementa Runnable, para luego ser tratada como un hilo, la única responsabilidad de esta clase es asignar vehículos a un viaje en estado solicitado.

Clases ClienteThread

El ClienteThread también implementa Runnable, esta clase simula el comportamiento de los clientes de la empresa, cada uno hace una cantidad máxima de pedidos.

Clases ChoferThread

El ChoferThread implementa Runnable, esta clase simula el comportamiento de los choferes de la empresa, cada uno hace una cantidad máxima de viajes. Si los choferes terminan de hacer el total de sus viajes, la simulación finalizara.

Recurso compartido

El recurso compartido es una clase llamada RecursoCompartido. Esta clase contiene todos los métodos synchronized de nuestro programa, este es el monitor que permite el acceso a dos arreglos: *ChoferesDisponibles* y *ClientesActivos*, en base a los cuales se determinará la ejecución de los correspondientes hilos.

El correcto funcionamiento del sistema depende de que el monitor regule el acceso a la zona crítica.

Esta clase también cuenta con 3 atributos estáticos que funcionan como banderas.

contClientesActivos lleva un registro de los clientes activos que hay en la simulación e impide que un ChoferThread quede a la espera de más viajes si no hay Clientes que los soliciten.

contChoferesActivos lleva un registro de los choferes activos que hay en la simulación e impide que un ClienteThread quede a la espera de que algún Chofer tome el pedido si no hay choferes “trabajando”.

contClientesHumanosActivos, idem a contClientesActivos, con la diferencia de que este contador refiere a los clientes que ingresaron mediante la interfaz usuario, y no elementos de la simulación (bots).

Conclusión

El proyecto final cumplió con las expectativas propuestas al comienzo de este.

Se logró simular el comportamiento de una aplicación de vehículos haciendo uso de ventanas y de la concurrencia.

Se logró un trabajo colaborativo respetuoso y logramos internalizar el uso correcto de los distintos patrones implementados.

El proyecto permitió que cada integrante del grupo obtuviera experiencia para el manejo de códigos entre muchas personas, en el uso del GitHub y en cómo trabajar colaborativamente.

Las dificultades encontradas fueron principalmente en el manejo de las versiones de Eclipse, ya que ante las distintas versiones y el hecho de ser una IDE relativamente nueva, se creaban conflictos al pasar de una a otra en el classpath. También el uso del GitHub trajo problemáticas al momento de hacer los push ya que al trabajar simultáneamente se creaba un conflicto entre las distintas versiones del código.

Con respecto al desarrollo del sistema su principal dificultad se presentó en el recurso compartido, que responsabilidades debía tener, como los hilos interactúan con él y que métodos debería sincronizar.

Como solución destacada podemos señalar el diseño e implementación del recurso compartido que gestiona choferes disponibles y viajes. Fue necesario distinguir los choferes y viajes ya que al ser los dos elementos a los cuales los múltiples hilos acceden, es importante gestionarlos de forma correcta. Además nos permite hacer más eficiente la búsqueda y recuperación de viajes y choferes, ya que de hacerlos con todos los choferes históricos de la empresa podríamos incurrir en búsquedas ineficientes y problemas de diseño a la hora de gestionarlos de forma concurrente.

Si bien hubo que realizar modificaciones al código original, se pudo aprovechar el buen diseño de la primera parte para delegar tareas que le son propias al sistema y realizar las operaciones necesarias desde el mismo recurso compartido. Esto nos hace pensar que el diseño que realizamos en la primera parte era adecuado debido a la posibilidad de haberlo extendido a un entorno concurrente sin necesidad de hacer grandes cambios en lo que ya teníamos y pudiendo aprovechar varias funcionalidades.

En el futuro seguiremos estudiando el manejo de GitHub, ya que la consideramos una herramienta muy poderosa, de la cual pudimos sacar mucho provecho a pesar de las dificultades que esta conlleva.