Programación III
Trabajo Práctico Integrador - Segunda parte
Grupo 7

de Aza, Paloma
Entrocassi, Regina
Gigliotti, Rocío
Salig, Tomás Francisco

2024

Indice

Introducción

El presente proyecto se basa en la simulación de un sistema para Radio-Taxi que gestiona clientes, choferes, vehículos y viajes. Este sistema administra un conjunto de vehículos de diferentes tipos y un listado de choferes, los cuales serán capaces de manejar cualquiera de los vehículos disponibles. Un cliente registrado en el sistema puede solicitar un pedido de viaje, el cual podrá ser aceptado o no según si es válido (coherente), si hay un vehículo adecuado disponible y un chofer disponible. Cuando el pedido es aceptado, se crea un viaje, el cual tendrá una evolución en el tiempo desde el momento en que es creado hasta su finalización.
La realización de este trabajo implica aplicar todos los conceptos y conocimientos adquiridos en la materia, permitiéndonos observar cómo se relacionan distintos componentes en un mismo programa y cómo interactúan entre sí. Además, en esta parte del trabajo, se incorporan temas como la concurrencia y la agregación de ventanas y elementos visuales,. Esto nos prepara para utilizar estas habilidades en el futuro, aplicando técnicas como el uso de Threads, ventanas, etc, que sin dudas utilizaremos a lo largo de la carrera.

Objetivos del trabajo

El objetivo principal de este trabajo fue obtener un entendimiento sólido de concurrencia y manejo de Threads, GUIs y persistencia. Para poder completar esta segunda parte de la entrega del Trabajo Global, además, debimos aprender a utilizar el patrón Observer/Observable, el patrón DAO-DTO y el patrón de diseño MVC.

Motivaciones

La realización del presente trabajo nos permitirá aprender a realizar un programa completo y complejo, con muchos elementos interactuando entre sí, en grupo. Ganaremos habilidades no solo técnicas, sino también interpersonales, al tener que trabajar de manera grupal para poder resolverlo.

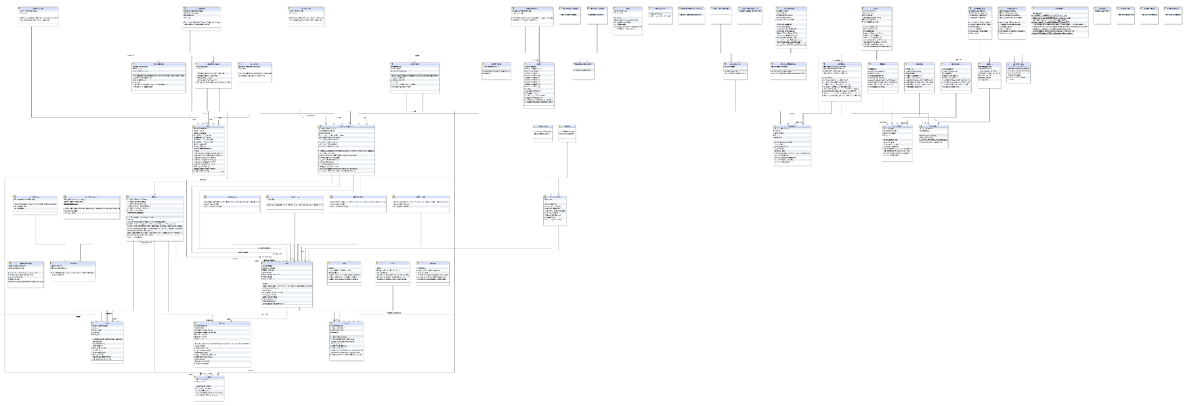
Desarrollo

Metodología

Para resolver el presente trabajo, partimos de lo entregado en la primera porción del mismo. Comenzamos el desarrollo de la segunda parte arreglando los errores marcados en la primera entrega (principalmente en la realización del Patrón Template y del Decorator). El orden en que decidimos encarar los nuevos agregados al trabajo fue, primero, encargarnos de la sincronización entre Threads. Luego, una vez la simulación funcione, pasamos a realizar las ventanas y vistas. Una vez esta segunda etapa estuvo completa, terminamos implementando la persistencia.

Diseño e implementación

Diagrama UML



(la imagen con resolución apropiada se encuentra en GitHub)

Diseño MVC

Modelo

Nuestro modelo es el **RecursoCompartido**. En el Recurso Compartido se encuentran todos los métodos de los Threads. El usuario (cliente humano) actuará como un client

Vista

En nuestro trabajo, contamos con dos vistas principales: **Login** y **Vista principal**.
La ventana de login cuenta con dos campos para ingresar texto (un textField para ingresar el usuario y un password Field para ingresar la contraseña) y dos botones: Un botón de Log In y un botón de Registrarse. La ventana se da cuenta cuando el usuario humano no ingreso valores en

los campos antes de darle click a los botones, y le avisa con una ventana emergente que ésto no es válido. Si el usuario ingresa valores en los campos, los botones tienen funcionamientos diferentes. El botón de Log In busca en el HashMap de usuarios si el usuario ingresado existe, y su contraseña concuerda con la ingresada. De ser así, lo deja ingresar. En otro caso, muestra una ventana emergente.
La ventana principal está separada en tres partes. En el panel izquierdo, deja que el usuario humano genere pedidos. En el panel central, muestra el avance de los pedidos del cliente humano. En el panel derecho, mediante un sistema de pestañas, muestra el avance de la simulación general, el de un Chofer en particular y el de un Cliente en particular.

Controlador

Contamos con dos controladores: **Controlador** (que controla la Ventana general) y **Controlador Login** (que controla la ventana de Login). Ambos Overridean el método actionPerformed de las ventanas, siendo quienes reaccionan cuando los botones de las ventanas son presionados.

Adaptación del modelo de 3 capas a MVC

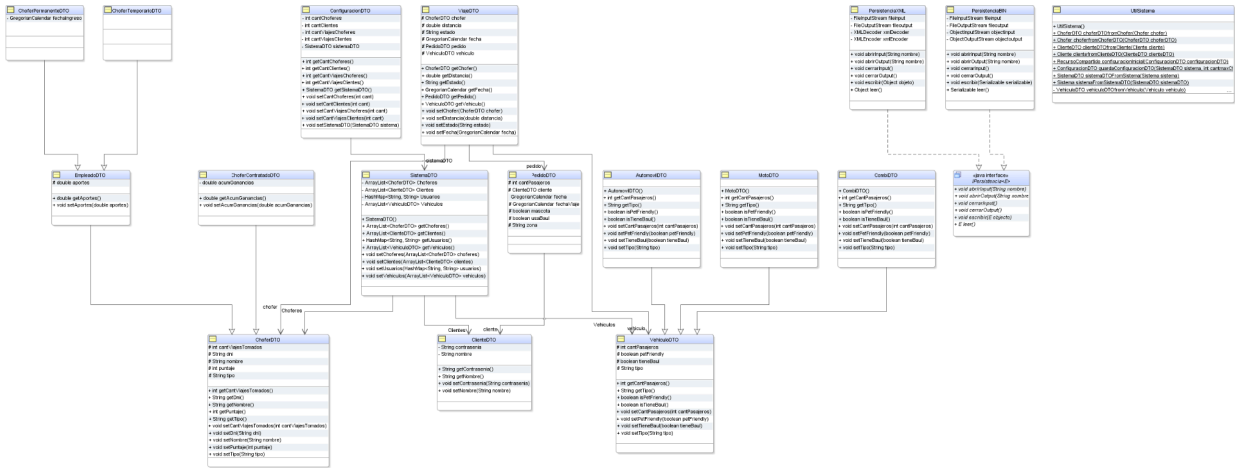
Adaptamos el modelo de 3 capas a MVC de la siguiente manera: guardamos las Vistas y los Controladores en la capa de presentación y el modelo en la capa de lógica de negocios. Decidimos esto ya que consideramos que las Vistas tan solo son relevantes en la Capa de Presentación.

Patrones utilizados

Aparte de los patrones utilizados en la primera parte, se implementaron tres patrones nuevos: DAO y DTO, Observer/Observable, y MVC.

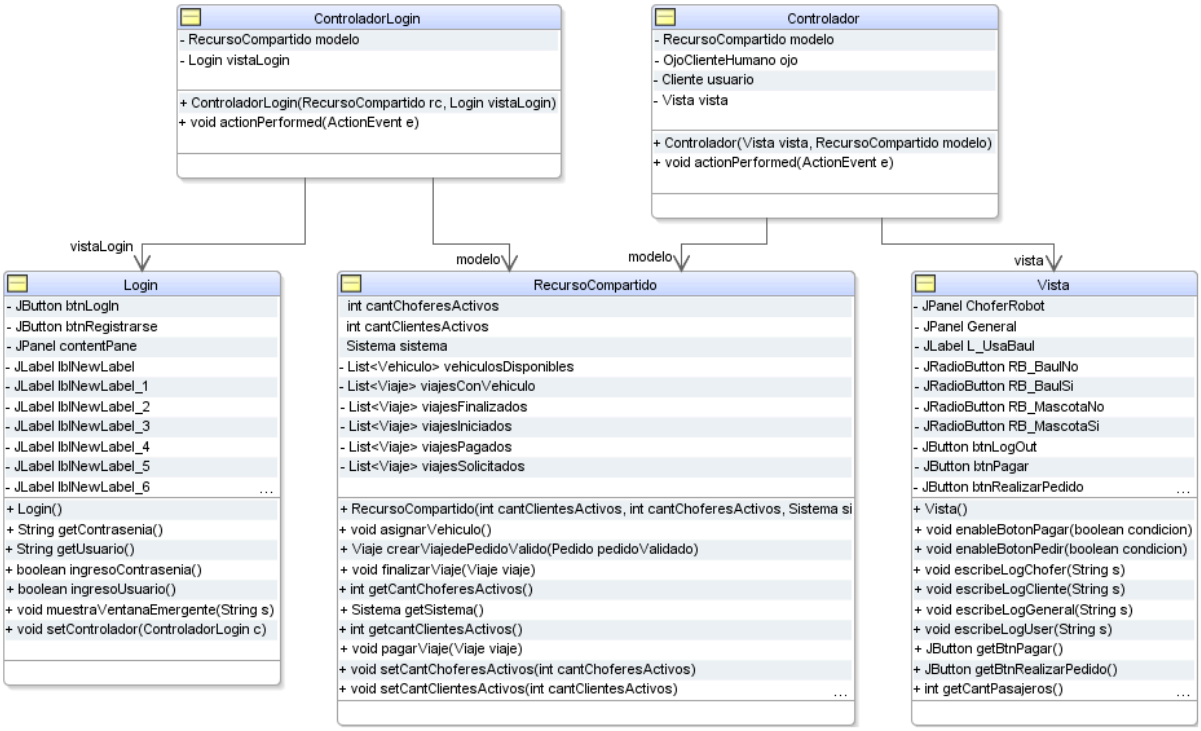
Patrón DAO y DTO:
Para poder persistir los datos del sistema y los datos iniciales de la configuración del mismo utilizamos el patrón DAO/DTO. El patrón DTO (Data Transfer Object) lo implementamos de manera de tener una clase DTO por cada clase cuyos datos queremos persistir, que pueda extraer y guardar determinados datos del sistema desde un archivo XML. Este patrón lo aplicamos mediante dos métodos main, SimulaciónInicial y SimulaciónFinal. En SimulaciónInicial nos encargamos de generar por primera vez los datos, y guardar estos mismos, que se cargaron previamente en el archivo XML de la clase correspondiente, esto ocurre a través de un objeto XMLDecoder, y de los setters y constructores correspondientes a cada clase. Durante SimulaciónFinal, a través de un XMLEncoder y getters, los datos del Sistema se cargan de el archivo XML (a través de una clase sistemaDTO) generado previamente, y luego una vez terminada la simulación, se vuelve a cargar el XML para su próximo uso. Notar que, a su vez,

SistemaDTO es parte de una clase ConfiguraciónDTO que contiene, además de el sistemaDTO, los parámetros iniciales con los que se configura el sistema antes de comenzar la simulación. Para utilizar el sistema, se debe iniciar el main de la clase“SimulacionInicial” una primera vez. En este momento, se puede crear un primer usuario. En todos los ingresos posteriores, se puede ingresar desde la clase “SimulacionFinal” y seguir creando usuarios o loguearte con usuarios previamente creados; los datos del sistema se persistirán, así como también los nuevos usuarios creados.



Patron MVC:

Como mencionamos anteriormente, el Recurso Compartido es el modelo, luego contamos con dos vistas (VistaGeneral y VistaLogIn) que conforman la parte Vista, y con dos controladores (ControladorLogin y controlador general), uno por cada vista. Ambos controladores se encargan de actualizar las vistas ante un cambio sucedido, así como las vistas también reaccionan a las acciones del usuario y el controlador se encarga de manejar los eventos.

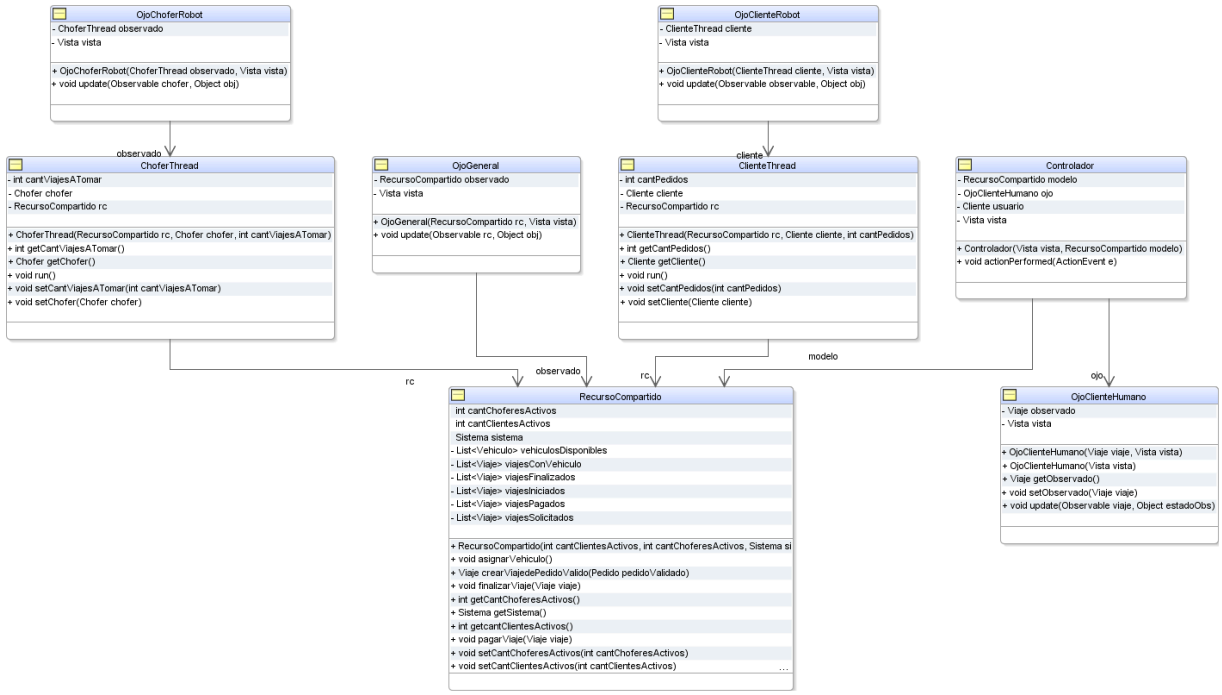


Patron Observer/Observable:

Para implementar el patrón Observer/Observable implementamos el uso de 4 “ojos”, estos son:

- Ojo cliente humano, este se encarga de notificarle a la vista todos los cambios de estado del viaje creado por el cliente humano. Para realizar esto, el ojo humano va a mantener a este viaje como su objeto observado, este se va a asignar en el momento donde el usuario presiona el botón “Pedir” y el pedido es validado, creando así un viaje solicitado. Este ojo va a seguir el proceso de todo el viaje, recibiendo notificaciones mediante el estado del viaje.
- Ojo cliente robot, este se encarga de notificarle a la vista los cambios de un cliente robot seteado manualmente desde el main. Este cliente robot va a ser un thread, y los cambios observados se van a dar en el método run(), que a su vez va a invocar a métodos sincronizados dentro de la clase recurso compartido.
- Ojo chofer, este se encarga de notificarle a la vista los cambios de un chofer robot seteado manualmente desde el main. Su funcionamiento es similar al ojo cliente robot, deberá observar a un thread chofer robot, y todos sus cambios se van a dar dentro del método run(), este thread también interactúa con los métodos sincronizados del recurso compartido.
- Ojo general, se encarga de notificarle a la vista todos los cambios que realizan los choferes, clientes y sistema sobre el recurso compartido. Su objeto observable será el

recurso compartido, el cual tiene métodos sincronizados que son accedidos por los choferes y clientes, por lo cual este ojo tiene la capacidad para seguir un viaje de principio a fin.



Diseño de clases.

Para esta segunda parte del trabajo, creamos varias clases nuevas.
Podemos comenzar hablando sobre las vistas y los controladores. Creamos las vistas utilizando el plugin WindowBuilder en Eclipse. Utilizando este plugin, pudimos diseñar las ventanas de manera gráfica, mientras que el código se generaba automáticamente. De todas maneras, nos parece importante notar éstas clases, debido a que cambiamos la manera en la que se generaba el código para las mismas. La generación de variables fueron de tipo Field, y el listener interface es implementado en la clase padre.
Aparte, creamos métodos en las clases a los que podrá acceder el controlador para poder cambiar éstas ventanas (enableando y disableando botones, por ejemplo). Por otro lado, los controladores deben ser quienes tengan los métodos de Action Listener.

Una de las clases más importantes es el **RecursoCompartido**. En ésta clase se encuentran todos los métodos synchronized a los que acceden las Threads. Es la clase principal en el manejo de la concurrencia.

Por ejemplo, el método **synchronized Viaje crearViajedePedidoValido** se implementa de la siguiente manera:

```
public synchronized Viaje crearViajedePedidoValido(Pedido pedidoValidado){
      Viaje viaje = null;
      viaje = sistema.crearViaje(pedidoValidado ,100);
      viajesSolicitados.add(viaje);
      viaje.setEstado("Solicitado");
      setChanged();
      notifyObservers("El viaje de "+
      viaje.getPedido().getCliente().getNombre()+" ha sido solicitado\n");
      notifyAll();
      return viaje;
}
```

Este método es utilizado por los **ClientesThreads** para iniciar un viaje a partir de un pedido que haya sido previamente validado. En el recurso compartido se crea el viaje, se lo agrega a una lista de **ViajesSolicitados** (que el **SistemaThread** está leyendo constantemente).
Todos los otros métodos utilizados por los Threads se pueden ver en **RecursoCompartido**.

Conclusiones

Cumplimiento de expectativas

Las principales expectativas que teníamos al comenzar el trabajo era ser capaces de realizar correctamente un trabajo que incluyera una parte visual de interacción con el usuario, y una parte de concurrencia. Desde un punto de vista académico, nos agradó poder ser capaces de crear un trabajo globalizador para el que tuvimos que poner a prueba nuestros conocimientos en éstas áreas. Desde un punto de vista de trabajo colaborativo, nuestras expectativas eran poder trabajar de manera exitosa en grupo. Pudimos cumplir ésta expectativa ya que, durante la realización del trabajo, pudimos trabajar grupalmente y participar en la resolución del trabajo.

Describir las dificultades encontradas y describir la solución.

Una de las principales dificultades que encontramos al momento de realizar el trabajo, resultó en la imposibilidad de testear la correcta funcionalidad de las cosas, ya que debido a la gran cantidad de elementos que estábamos manejando al mismo tiempo, teníamos que tener ya gran parte de las cosas definidas para poder correr el programa, y a partir de allí, rastrear los errores que ocurrieron. Esto fue solucionado agregando todos los métodos necesarios y colocando salidas por pantalla (por consola) para guiarnos en lo que debería ser el correcto funcionamiento de la simulación.

Otra dificultad que surgió fue al momento de pensar y crear los métodos sincronizados para el recurso compartido. Si bien, sabíamos cuáles eran los métodos a los cuales podían acceder los hilos, a veces resultaba confuso pensar cómo deberían comportarse, sobre todo al momento de verificar que funcionaban de manera correcta, y que ocupando de a uno por vez el **Recurso Compartido**. Al mismo tiempo, debíamos chequear que cada hilo terminara de ejecutar su método run(), y que pueda acceder a los métodos correspondientes. Esto también se resolvió colocando salidas por pantalla que indiquen la actividad de cada hilo de ejecución. Cabe aclarar que esta metodología solo se utilizó al momento de corregir errores, y que no se deben mostrar al momento de correr el programa.

El poder persistir los datos del sistema nos representó un trabajo muy grande, porque nuevamente, chequear que funcionara de manera correcta era complicado. Al descubrir que el archivo XML no se escribía correctamente, tuvimos que exportar los paquetes de Capa de Lógica de Negocios para que la capa de Persistencia los reconociera. Luego, accediendo al

contenido del archivo XML, y corriendo la parte del programa donde se despersisten los datos, pudimos comprobar que se persistían los datos correctamente.

Solución destacada

Consideramos que nuestro mayor logro fue resolver el problema globalmente, con esto nos referimos a que para que el programa funcionara correctamente, debimos lograr sincronizar la simulación de threads, las vistas, la persistencia y el trabajo del usuario.
Por ejemplo, si queríamos loguear y registrar usuarios, para luego ingresar a la vista general, debemos diseñar las ventanas, con los controladores de cada ventana, la persistencia (para guardar aquellos usuarios previamente registrados) y con los Threads (para iniciar la simulación una vez abierta la ventana).
Por lo tanto, nuestra "solución destacada" no se trata de una parte de cierta parte del código, sino de haber podido vincular todas las partes del proyecto para que este funcione correctamente.

Aprendizaje

Consideramos que lo que haríamos diferente si tuviéramos que volver a realizar esta experiencia es realizar una GUI distinta. Podríamos averiguar la manera de hacer una interfaz de usuario más moderna o que nos sea más agradable.
Consideramos que el orden en que encaramos el trabajo fue el apropiado. Poder encargarnos en un primer momento de que la concurrencia funcione correctamente nos permitió poder abocarnos a la parte visual posteriormente. Una vez todo estuvo funcionando correctamente, proseguimos a encargarnos de analizar la parte de persistencia.
Ahora cambiando el eje de la discusión, consideramos, igual que en la primera entrega, que lo más importante que pudimos aprender fue a trabajar en grupo de manera efectiva. Desarrollar habilidades interpersonales y poder trabajar con otros es fundamental para nuestra futura vida profesional, y es necesario que obtengamos estas habilidades durante la carrera.