Informe Trabajo Práctico Final

Primera parte

Programación III - Facultad de Ingeniería UNMDP

Profesores:

Guccione, Leonel;

Lazzurri, Guillermo;

Gellon, Ivonne

Integrantes:

de Aza, Paloma; Entrocassi, Regina; Gigliotti, Rocio; Salig Tomás

Fecha de entrega: 05/05/2024

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 implementación de este sistema se basa en el paradigma de la Programación Orientada a Objetos (POO) aprendido durante este curso. Conceptos a destacar son el polimorfismo, la herencia, el principio de Liskov, el manejo de excepciones y el uso de diversos patrones de diseño, cuya implementación será detallada en el presente informe.

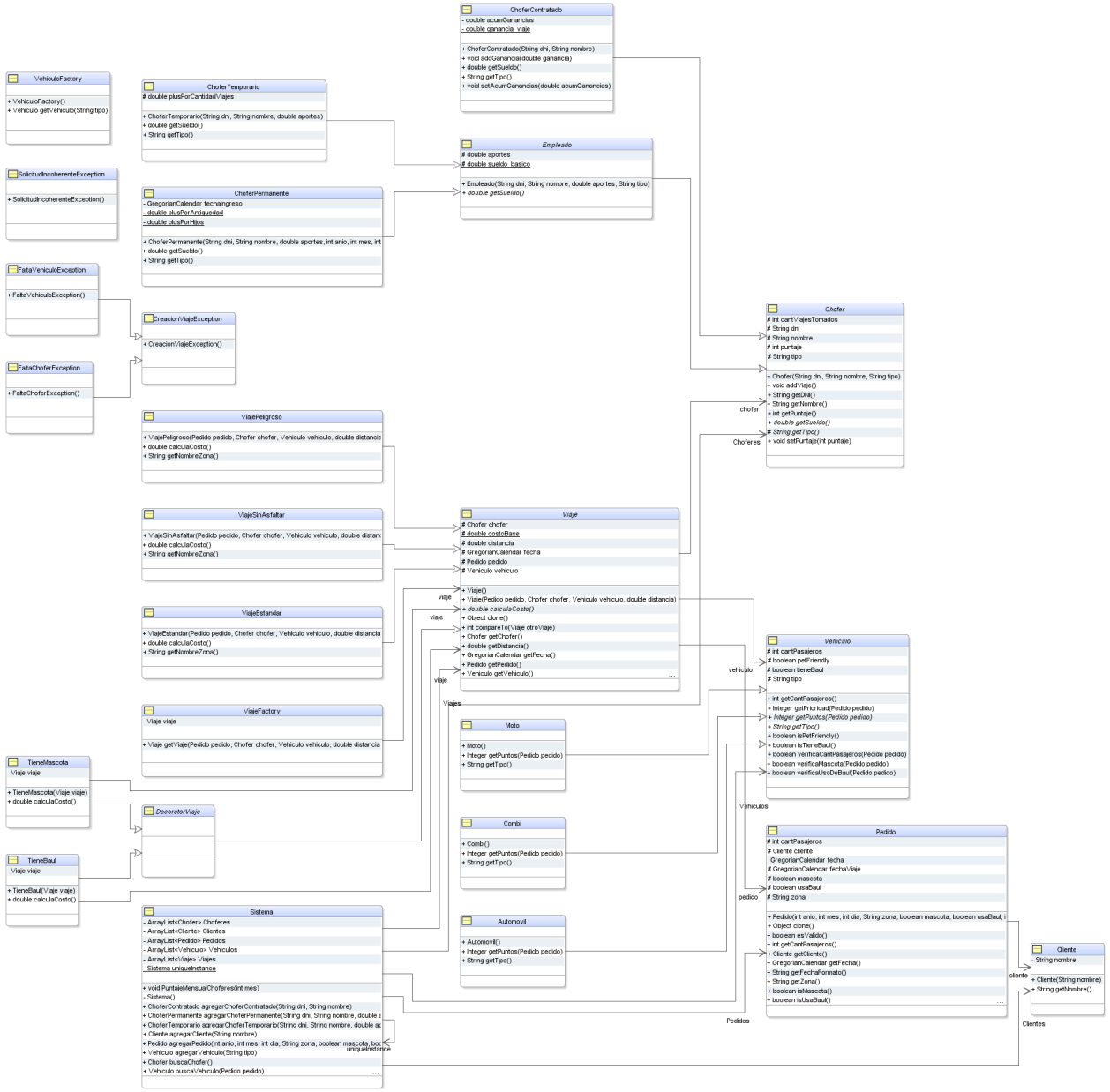
Contrato

En esta primera parte **Sistema** debe ser capaz de crear tanto **Clientes**, como **Choferes** y **Vehículos**. En el momento en que **Cliente** crea un **Pedido**, se debe verificar que la solicitud sea válida. La cantidad de pasajeros debe encontrarse entre uno y diez, y de ser mayor a cuatro no debe solicitar llevar una mascota. Todos estos datos son verificados en el método **EsVálido()** que lanza la excepción **solicitudIncoherenteExcepcion** de no cumplirse.

Cabe aclarar que, en esta primera entrega, todos los datos que se les proporcionan a los constructores de las clases son ingresados por el programador en la clase **Prueba**. Es decir, se deben ”hardcodear” ya teniendo una noción de cuáles son los datos que causarán o no un lanzamiento de excepciones. Más adelante, estos datos serán ingresados por un usuario por lo cual es importante que se especifique en el contrato qué se espera que se ingrese para el correcto funcionamiento del sistema.

De todas formas, en el Javadoc se aclaran las precondiciones para aquellos métodos en los cuales se ingresaba un **Pedido** como parámetro. También se incluye una breve descripción de los métodos y las clases, explicando su funcionalidad y el tipo de retorno. El Javadoc del proyecto se encuentra en el repositorio de GitHub.

Diagrama de clases



La jerarquía se muestra de izquierda a derecha.

(Un archivo png con mayor resolución se encuentra en el repositorio de GitHub)

Patrones de diseño utilizados

Singleton

- Aplicamos el patrón Singleton sobre la clase **Sistema**, de manera que obtuvimos una única instancia de esta clase que controla todo el funcionamiento del sistema, incluyendo la instanciación de Objetos, la generación de reportes, etcétera. Todos los métodos utilizados en la clase **Prueba** son definidos en Sistema.

```
public static Sistema getInstance() {
	if (uniqueInstance == null) {
		uniqueInstance = new Sistema();
	}
	return uniqueInstance;
}
```

Factory

- Para los **vehículos**: Utilizamos el patrón para crear los vehículos del sistema y así poder instanciar distintos tipos de vehículo, es decir, poder crear vehículos de tipo **Automóvil**, **Moto** y **Combi**. Para su implementación creamos las clases concretas **Automóvil**, **Moto** y **Combi** de forma que éstas extiendan de la clase abstracta **Vehículo** y a través de otra clase **VehículoFactory**, mediante un “árbol” de if/else la Factory decide qué tipo de **Vehículo** concreto instanciar.

- Para los **viajes** (encapsulando el decorator): El **ViajeFactory** se utiliza para poder asegurar que el viaje se decora correctamente. Es común utilizar el Patrón Decorator junto con el Patrón Factory para asegurar el correcto encapsulamiento y funcionamiento del objeto decorado. En nuestro caso, la factory crea un viaje de un cierto tipo (según la zona), y luego lo decora según las especificaciones del pedido.

Decorator

- Para viajes: El patrón decorator es utilizado para decorar nuestro objeto **viaje** dependiendo de si usa baúl o si tiene mascotas. Para ello, se creó la clase abstracta **DecoratorViaje**, que es una subclase de la clase abstracta **Viaje**. Los decorators concretos son **TieneMascota** y **TieneBaul**, y estos decorators sirven para modificar el método **CalculaCosto()** de **Viaje**.

Template

- Para **vehículo**: El patrón define el orden en el que se verifican las condiciones del vehículo para ser apto para el pedido, finalizando con el cálculo de **getPuntos()**. De esta manera, se obtiene la prioridad que tiene un cierto vehículo para un pedido en particular.

```
public Integer getPrioridad(Pedido pedido) {
	boolean verificaCant = verificaCantPasajeros(pedido);
	boolean verificaBaul = verificaUsoDeBaul(pedido);
	boolean verificaMascota = verificaMascota(pedido);
	if (verificaCant && verificaBaul && verificaMascota){
		return getPuntos(pedido);
	} else {
		return null;
	}
}
```

- Para **viajes**: En la creación de viajes, aplicamos el patrón Template para establecer la secuencia que debe seguir el método para poder generar un viaje. El método debe, una vez recibido un pedido coherente, verificar si hay algún vehículo disponible que pueda tomarlo y elegir el de máxima prioridad, luego encontrar un chofer disponible, y como paso final, si no se ha lanzado ninguna excepción, efectivamente crear el viaje.

Facade

- Nuestra clase **Sistema**, que a su vez funciona como nuestra “fachada”, nos proporciona una interfaz única y simple, que permite ocultar la complejidad al usuario y ofrece una forma más fácil de interactuar con los demás objetos. El **Facade Sistema** encapsula y gestiona la funcionalidad interna y las interacciones entre objetos.

Modificaciones

Una de las mayores modificaciones que debimos implementar fue el eliminar la interfaz **IViaje**. Nos dimos cuenta de que, para aplicar el patrón Decorator junto con el Patrón Factory para decorar y crear los viajes, nos resultó más conveniente utilizar una clase abstracta **Viaje** de la que heredaran tanto clases concretas **ViajeZonaPeligrosa**, **ViajeEstandar**, **ViajeZona**, como también una clase abstracta **DecoratorViaje**.

Otra modificación importante que realizamos fue implementar métodos en **Sistema** para instanciar **Choferes**, **Vehículos**, **Pedidos** y **Viajes**, en vez de instanciarlos directamente en el

método main de **Prueba**. Tomamos esta decisión debido a que nos pareció correcto que fuera **Sistema** quien se encargara de instanciar estos objetos, para respetar el patrón **Facade**.

Otra de las responsabilidades de la clase **Sistema** fue generar los reportes pedidos. Debido a que la clase **Sistema** no debería imprimir nada, los métodos de reporte y listados devuelven ArrayLists con los datos pedidos. Luego, en prueba se imprimen las listas de reportes correctamente formateadas.

Finalmente, decidimos crear tres Exceptions: **SolicitudIncoherenteException**, que lanzamos cuando un pedido no es coherente (el cliente no ha leído la cartilla de especificaciones del sistema), **FaltaChoferException** (cuando no hay ningún chofer que pueda tomar el pedido) y **FaltaVehiculoException** (cuando no hay ningún vehículo que pueda tomar el pedido).
Las dos últimas excepciones se heredan de la clase **CreacionViajeException** debido a que si no se encuentra un chofer o un vehículo, no se puede crear un viaje.

Mayores desafíos

Uno de los mayores desafíos que tuvimos, y en el que empleamos más tiempo, fue poder implementar correctamente el patrón Decorator con el patrón Factory para poder crear **viajes** y calcular el costo de los mismos. Nos sirvió basarnos en las diapositivas de las clases así como también en los libros: *Design Patterns. Elements of reusable object oriented software* y *Head First Design Patterns*.

Aparte de esto, la implementación de los atributos de fecha para algunas de las clases que lo requerían también supuso un desafío. Finalmente optamos por utilizar las librerías **GregorianCalendar** para la instanciación de fechas y para el cálculo de tiempo entre una fecha y otra cuando era necesario, y **SimpleDateFormat** para mostrar las fechas en un formato adecuado.

Partes favoritas del desarrollo

Nos sorprendió como los mayores desafíos del desarrollo terminaron estando relacionados con nuestras partes favoritas del mismo. Como ya mencionamos, uno de los mayores desafíos del trabajo fue poder implementar correctamente el Patrón Decorator junto con el Patrón Factory para la creación de los viajes. En un cierto punto, creamos un pequeño proyecto en Java que simulara tan solo esta parte del desarrollo, para poder encontrar el error que estábamos teniendo en el diseño. Lo testeamos, y nos resultaba imposible poder ver el punto de error, hasta que, luego de ya pasada casi una hora, ¡nos dimos cuenta! Luego de esto pudimos arreglar el funcionamiento de estos patrones, y ésto nos resultó muy satisfactorio.

Otra de las partes que nos agradó mucho fue, una vez definidas todas las clases y métodos, poder crear distintos casos en nuestro método main en Prueba y rastrear los errores cuando algo no se comportaba como debía. Nos conectamos de manera virtual y, mientras uno de nosotros compartía su pantalla con el proyecto en Eclipse, los otros intentábamos encontrar los errores, proponer donde poner un breakpoint, que variables observar, etcétera. En general, éste terminó siendo un proceso bastante entretenido.
Además de esto, también nos gusto buscar bibliografía sobre estos temas, lo cual nos sirvió para entender mejor los conceptos y patrones de diseño de la POO, y expandir nuestro conocimiento sobre los mismos.
Otra parte que nos agradó de éste trabajo es que debimos aprender a manejar diversos entornos de desarrollo para poder realizarlo. Consideramos que ganamos la capacidad de trabajar con herramientas nuevas, y creemos que esto será de mucha utilidad en el futuro.
Por último, una de nuestras partes favoritas fue poder trabajar en grupo. Pudimos repartirnos las tareas, organizar los tiempos, y en general el trabajo fue más agradable al poder trabajar todos juntos.