Grupo 05
Integrantes : Arce Ezequiel, Esposito Cristian, Galarco Barrios Angelina, Nieto Iván Ezequiel.
Programación Ⅲ - Ingeniería en Informática
Facultad de Ingeniería

Subí que te llevo

6 de Mayo de 2024

RESUMEN

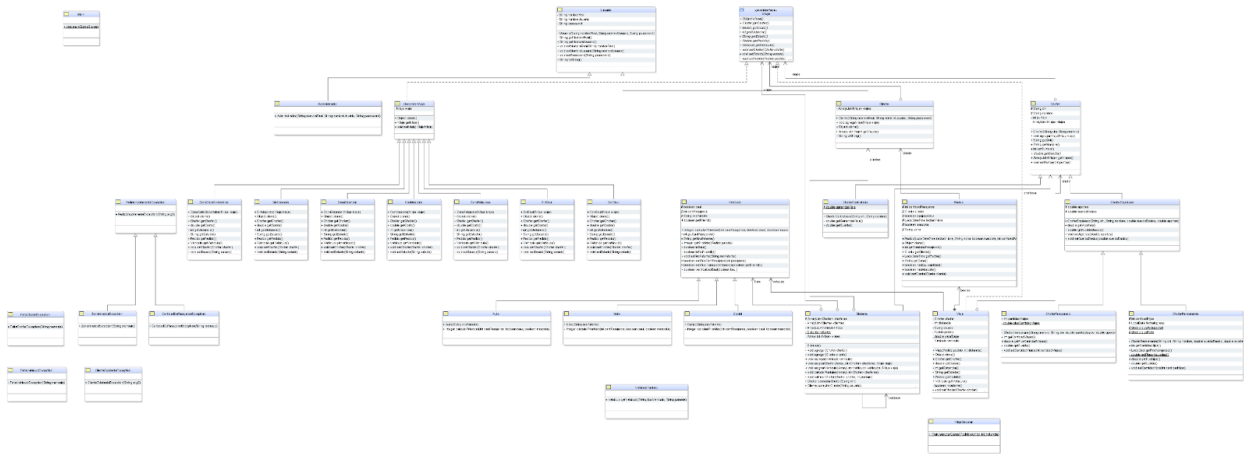
El presente proyecto consiste en un sistema para una empresa de transporte de pasajeros implementado en Java. El sistema se encarga de gestionar la información de la empresa y tiene dos categorías de usuarios, clientes y un único administrador.
La empresa cuenta con una flota de vehículos, un conjunto de choferes y un conjunto de clientes registrados con los cuales opera.
Los clientes pueden solicitar un viaje y el sistema analiza la posibilidad de concretar ese pedido, tener acceso a todos los viajes que realizan y también pueden calificar a los choferes con los que viajan.
El administrador es el encargado de la gestión de clientes,choferes y vehículos. Se ocupa de las altas de nuevos clientes, vehículos y choferes, como también puede acceder a los registros de clientes, choferes, vehículos y viajes que tiene la empresa.
La flota de vehículos de la empresa cuenta con tres tipos distintos de vehículos, motos, automóviles y combis, cada uno con determinadas prestaciones. El sistema se encarga de asignar el vehículo más adecuado para cada viaje a partir de los requisitos del pedido y de las prestaciones de los vehículos disponibles.
Los choferes son otro elemento de la empresa y existen tres tipos de choferes, permanentes, contratados y temporarios.
Al momento de realizar un pedido para solicitar un viaje, el cliente llena un formulario con la siguiente información: fecha, zona (estándar, sin asfaltar, peligrosa), si tiene mascota o no, si requiere baúl y la cantidad de pasajeros. Luego a partir de toda esta información el sistema analiza si se puede realizar el viaje y asigna el vehículo más adecuado para el mismo, y en caso de no poder realizarse informa sobre el motivo.

DISEÑO DEL SISTEMA

El sistema se implementó basado en una arquitectura en capas. Específicamente en el modelo de Arquitectura en 3 capas:

- Capa de modelo: aquella encargada de almacenar los datos (Usuario,Chofer,Vehiculo,Viaje y Pedido);
- Capa de negocio: Validaciones complejas, entidades y procesos. En esta primera parte es el Sistema en su totalidad.
- Capa de Presentación: Aquella que muestra (presenta) la información en pantalla e interactúa con el usuario. Implementada en el bajo el patrón facade.

Se diagramó de la siguiente manera:



En el repositorio:
Se decidió respetar una programación basada en contratos, las descripciones de cada clase con sus respectivos métodos y contratos se pueden ver en el javadoc que se encuentra en el repositorio:

Se utilizó MAVEN para trabajar en conjunto con las distintas IDEs.

IMPLEMENTACIÓN

Se implementó el uso de distintos patrones, los cuales fueron Template, Decorator, Factory, Singleton y FACADE.

El patrón Template se utilizó dentro del cálculo de prioridades de los vehículos, esto se visualiza ya que el Template es una sucesión de métodos, en nuestro caso esto ocurre dentro de un if para evaluar si debemos calcular la prioridad del vehículo, si este puede cumplir el pedido se calcula su prioridad y si no lo puede cumpler devuelve null el método.

El patrón Factory se utilizó en dos partes del código, en la parte de Vehículo y en la de Viaje. El Factory se utilizó en el main para la creación de vehículos de distintos tipos(Moto, Auto y Combi), solamente era necesario pasarle como parámetro la patente y el tipo de vehículo que se quería crear. Su implementación dentro de la clase VehiculoFactory es la de una encadenación de if.

La relación entre el Factory y la clase viaje se debió a que se tenía que aplicar el patrón Decorator sobre esta y para poner las capas vimos la necesidad de aplicar este patrón para poder solucionar el problema de añadir las capas del Decorator, su implementación se encuentra dentro de la clase ViajeDecorar, a diferencia del Factory del vehículo este no son if encadenados sino que son if secuenciales, porque las capas se añaden una detrás de la otra, en su contraparte de vehiculo se crea una moto o un auto, no ambos al mismo tiempo.

El patrón Singleton se utilizó tanto para la clase Administrador y Sistema, en ambos se utilizó debido a que solo debe haber una instancia de esas clases en el programa, y se logró mediante el ocultamiento de ambos constructores y que la única manera de acceder a la instancias de ese clases sea mediante un método que controla si ya existe una instancia.

El patrón FACADE se implementó dentro de la capa de negocio, sería la clase sistema, todas los mensajes desde la capa de presentación se hacen a través del sistema, de esta manera encapsulamos lo que se le muestra el usuario, y le ponemos límites a lo que el usuario puede hacer dentro del programa.

El patrón Decorator se implementó únicamente para la clase viaje, se utilizó para agregar capas al viaje y no tener una explocion de clases con todos los tipos distintos de viajes, principalmente su función en el código se limita para el cálculo del costo del viaje, ya que este varía con las especificaciones de cada pedido, por ejemplo, el costo de un pedido que pasa por una zona estándar es menor al que pasa por una zona peligrosa, por lo que dependiendo al pedido que se realizó se fueron añadiendo capas al viaje para poder calcular el costo total que debería pagar el cliente.

Las funcionalidades del cliente y del administrador en nuestro programa se limitaron a estar dentro del sistema, es decir, que el programa no hace diferencia en quién está utilizando el programa, esto ocasiona que cualquier usuario puede utilizar todas las funcionalidad públicas del sistema, este problema se debió a que sería muy complicado y quedaría un código muy complicado de leer si se implementa una condición que evalúe si es un cliente o un administrador el que está usando el sistema, porque cada método debería preguntar qué tipo de usuario es, luego esto en un futuro será controlado a partir de una ventana de ingreso de usuario.

El nombre de la clase ViajeDecorar y su diferencia a DecoratorViaje es que ViajeDecorar es una clase que trata de hacer un Factory pero como no son if encadenados no queríamos generar una confusión poniendo el nombre de FactoryViaje, así que le pusimos ese nombre para evitar malentendidos. Mientras que DecoratorViaje es una clase abstracta que se utiliza para poder aplicar el patrón Decorator, su uso es para el encapsulamiento.

```java
package modelo;

public class ViajeDecorar { //Hice el metodo static para no inicializar igual se lo puede sacar e instanciar ViajeFactory en sistema
    /**
     * Añade las capas para encapsular Viaje<br>
     * <b>Pre: </b> El pedido debe ser distinto de null y debio haber sido validado, y la distancia debe ser mayor a cero<br>
     * <b>Post: </b> <br> Retorna la instancia de viaje con las capas del decorator
     * @param pedido Instancia del pedido realizador por el cliente
     * @param distancia Distancia a recorrer del viaje
     * @return Instancia del viaje
     */
    public static IViaje agregarCapas(Pedido pedido, int distancia) {
        assert pedido!=null: "El pedido debe ser distinto de null";
        assert distancia>0: "La distancia debe ser mayor a cero";
        IViaje respuesta=new Viaje(pedido, distancia);
        if(pedido.hasMascota())
            respuesta=new ConMascota(respuesta);
        else
            respuesta=new SinMascota(respuesta);
        if(pedido.hasEquipajeBaul())
            respuesta=new ConBaul(respuesta);
        else
            respuesta=new SinBaul(respuesta);
        if(pedido.getZona().equalsIgnoreCase("Peligrosa"))
            respuesta=new ZonaPeligrosa(respuesta);
        else if(pedido.getZona().equalsIgnoreCase("Estandar"))
            respuesta=new ZonaEstandar(respuesta);
        else
            respuesta=new ZonaCalleSinAsfaltar(respuesta);
        assert respuesta!=null: "Debe retornar una instancia de viaje";
        return respuesta;
    }

}
```

DESAFÍOS Y METAS ALCANZADAS

Principales desafíos que abordamos

1. La incorporación de GitHub como herramienta de trabajo. En un inicio fué complejo trabajar con Eclipse, GitHub y JDeveloper en simultáneo. Luego de haber finalizado esta primera entrega estamos muy satisfechos con la utilidad de las mismas.
2. El seguimiento del programa: A la hora de testear el correcto uso de las funcionalidades encontramos fallas que fuimos resolviendo clase a clase.
3. Desarrollar la lista de vehículos: Un problema que tuvimos fue que para el ArrayList<vehiculo> que se encuentra en Sistema, que representa la flota de vehículos que tiene la empresa, en un principio quisimos considerarlo como un HashMap, donde la clave era el número de patente de cada vehículo, pero teníamos problemas a la hora de recorrerlo y nos decidimos por modificarlo a un ArrayList porque no podíamos manejarlo como una cola, ya que a la hora de asignarle un vehículo a un viaje éste debe ser removido de la lista y cuando finaliza dicho viaje debe ser agregado a dicha lista en la última posición, y esto es algo que si utilizábamos un HashMap no íbamos a poder hacer.
4. Desarrollar el método clone() para el listado de viajes: El problema con el que nos encontramos al momento de clonar un viaje fue que perdimos las capas decoradas durante la clonación, ya que en el código desarrollado las capas se van desarmando para poder clonarse pero nunca vuelven a armarse.

Metas Alcanzadas

Creemos que fuimos prácticos a la hora de encarar este proyecto. Primeramente haciendo el diagrama de clases y luego codificando las funcionalidades. Esto nos permitió tener claro cómo se iban a integrar las clases y facilitó la codificación de las funcionalidades.