UNMdP – F.I. – Ingeniería en Informática

Prog. III

Trabajo Final : “Subí que te llevo”

Primera Parte

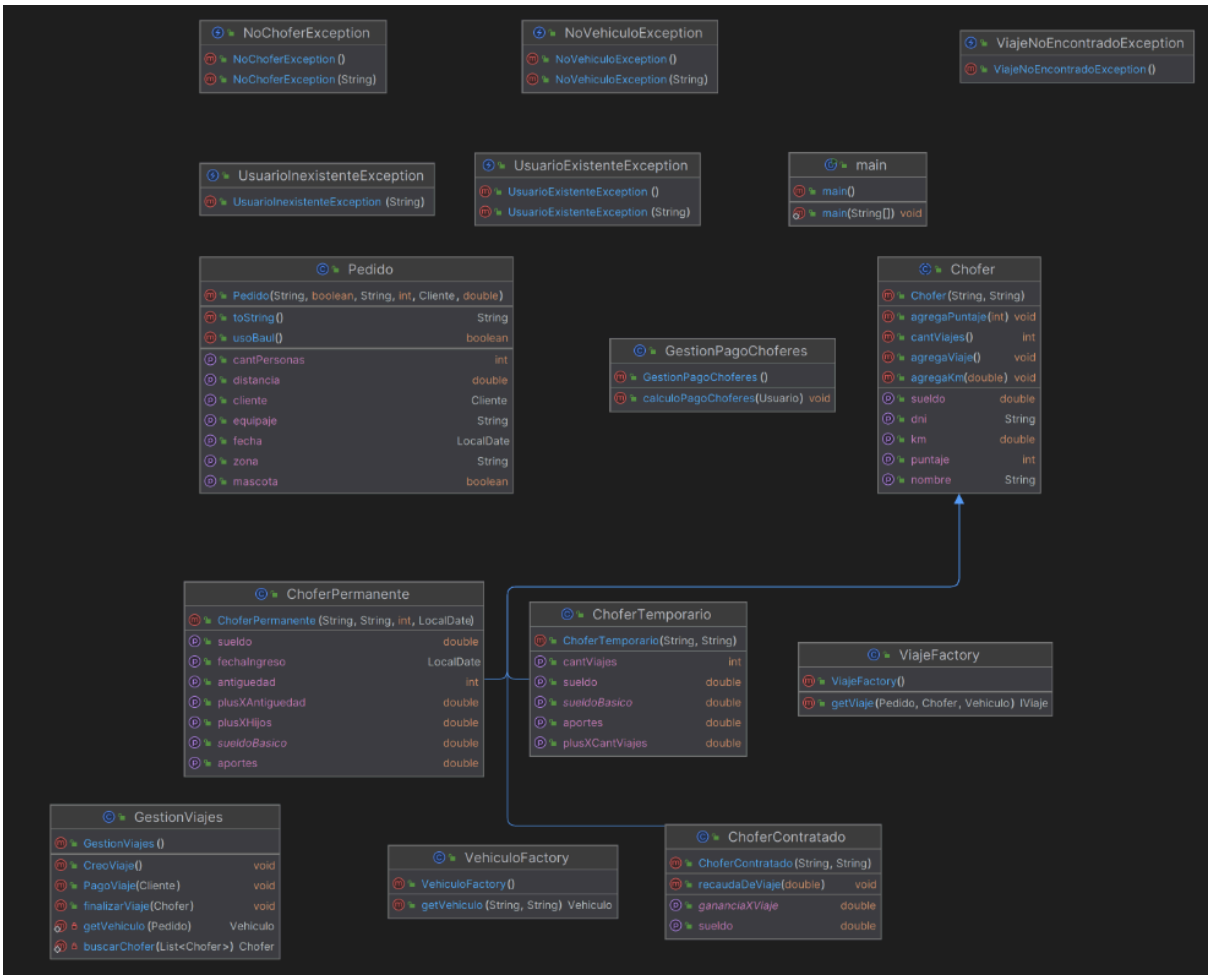
Grupo 8

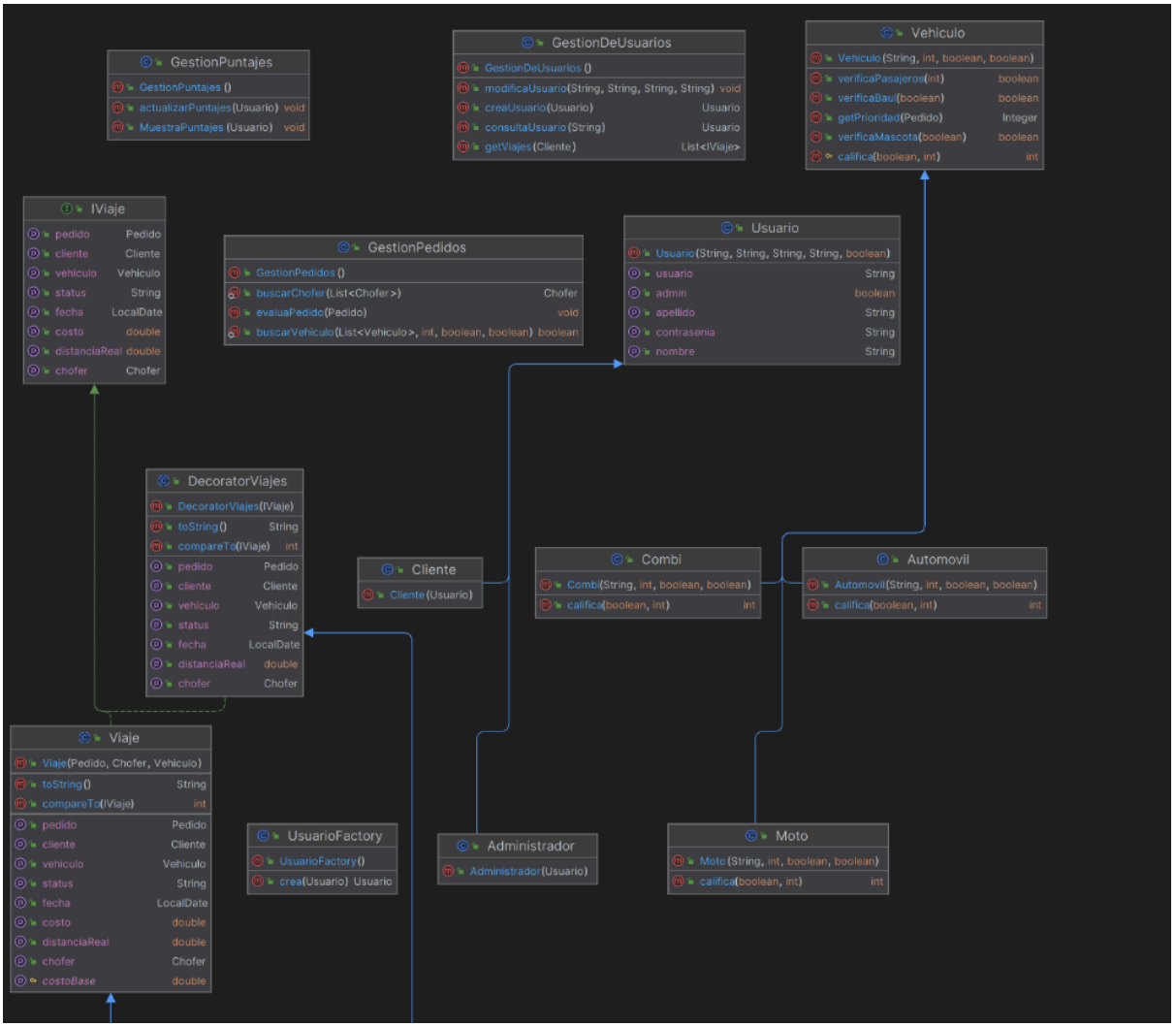
Integrantes:

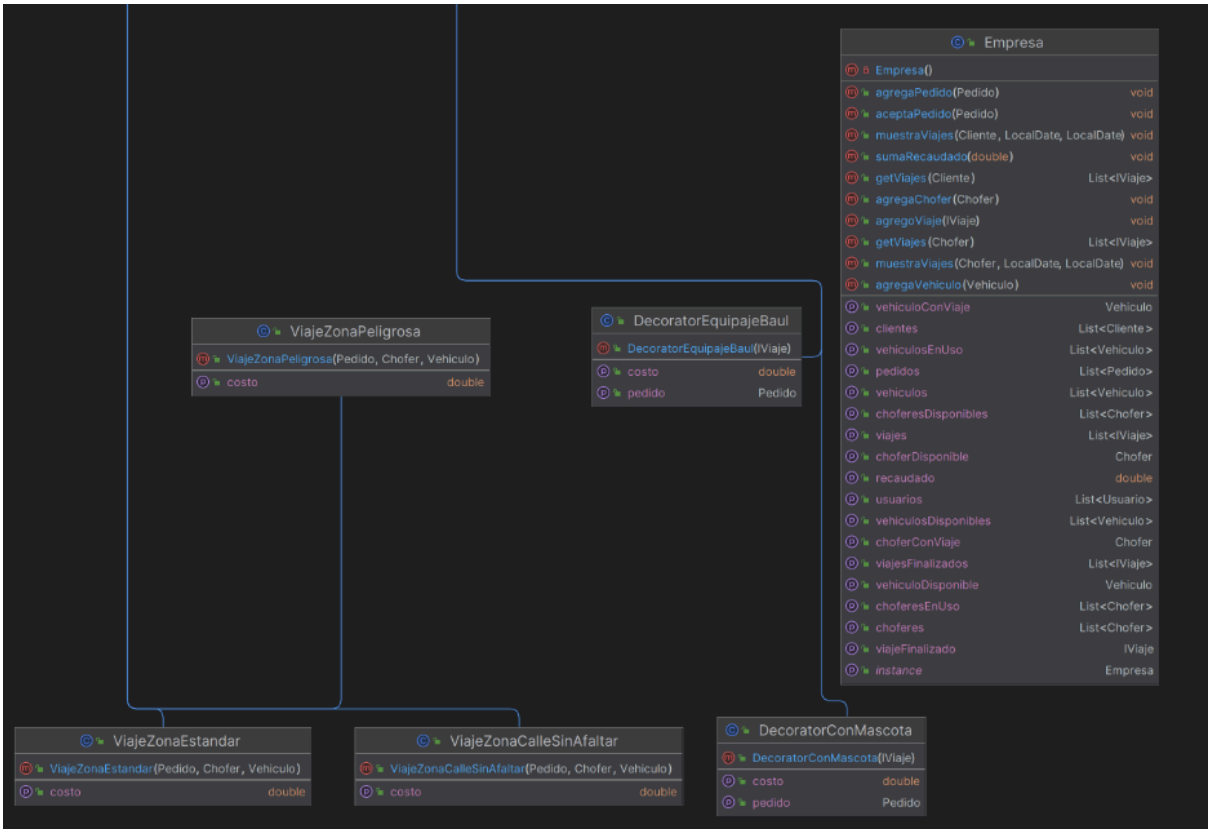
- Sofia Isabella Palladino
- Bautista Rodriguez
- Agustin Chazarreta
- Thiago Parise

Fecha entrega: 05/05/2024

UML


NoChoferException
NoChoferException ()
NoChoferException (String)
NoVehiculoException
NoVehiculoException ()
NoVehiculoException (String)
ViajeNoEncontradoException
ViajeNoEncontradoException ()
UsuarioInexistenteException
UsuarioInexistenteException (String)
UsuarioExistenteException
UsuarioExistenteException ()
UsuarioExistenteException (String)
main
main()
main(String[]) void
Pedido
Pedido(String, boolean, String, int, Cliente, double)
toString() String
usoBaul() boolean
cantPersonas int
distancia double
cliente Cliente
equipaje String
fecha LocalDate
zona String
mascota boolean
GestionPagoChoferes
GestionPagoChoferes ()
calculoPagoChoferes(Usuario) void
Chofer
Chofer(String, String)
agregaPuntaje(int) void
cantViajes() int
agregaViaje() void
agregaKm(double) void
sueldo double
dni String
km double
puntaje int
nombre String
ChoferPermanente
ChoferPermanente(String, String, int, LocalDate)
sueldo double
fechaIngreso LocalDate
antiguedad int
plusXAntiguedad double
plusXHijos double
sueldoBasico double
aportes double
ChoferTemporario
ChoferTemporario(String, String)
cantViajes int
sueldo double
sueldoBasico double
aportes double
plusXCantViajes double
ViajeFactory
ViajeFactory()
getViaje(Pedido, Chofer, Vehiculo) IViaje
GestionViajes
GestionViajes ()
CreoViaje() void
PagoViaje(Cliente) void
finalizarViaje(Chofer) void
getVehiculo(Pedido) Vehiculo
buscarChofer(List<Chofer>) Chofer
VehiculoFactory
VehiculoFactory()
getVehiculo(String, String) Vehiculo
ChoferContratado
ChoferContratado(String, String)
recaudaDeViaje(double) void
gananciaXViaje double
sueldo double


GestionPuntajes
GestionPuntajes ()
actualizarPuntajes(Usuario) void
MuestraPuntajes (Usuario) void
GestionDeUsuarios
GestionDeUsuarios ()
modificaUsuario(String, String, String, String) void
creaUsuario(Usuario) Usuario
consultaUsuario(String) Usuario
getViajes(Cliente) List<IViaje>
Vehiculo
Vehiculo(String, int, boolean, boolean)
verificaPasajeros(int) boolean
verificaBaul(boolean) boolean
getPrioridad(Pedido) Integer
verificaMascota(boolean) boolean
califica(boolean, int) int
IViaje
pedido Pedido
cliente Cliente
vehiculo Vehiculo
status String
fecha LocalDate
costo double
distanciaReal double
chofer Chofer
GestionPedidos
GestionPedidos ()
buscarChofer(List<Chofer>) Chofer
evaluaPedido(Pedido) void
buscarVehiculo(List<Vehiculo>, int, boolean, boolean) boolean
Usuario
Usuario(String, String, String, String, boolean)
usuario String
admin boolean
apellido String
contrasenia String
nombre String
DecoratorViajes
DecoratorViajes(IViaje)
toString() String
compareTo(IViaje) int
pedido Pedido
cliente Cliente
vehiculo Vehiculo
status String
fecha LocalDate
distanciaReal double
chofer Chofer
Cliente
Cliente(Usuario)
Combi
Combi(String, int, boolean, boolean)
califica(boolean, int) int
Automovil
Automovil(String, int, boolean, boolean)
califica(boolean, int) int
Viaje
Viaje(Pedido, Chofer, Vehiculo)
toString() String
compareTo(IViaje) int
pedido Pedido
cliente Cliente
vehiculo Vehiculo
status String
fecha LocalDate
costo double
distanciaReal double
chofer Chofer
costoBase double
UsuarioFactory
UsuarioFactory()
crea(Usuario) Usuario
Administrador
Administrador(Usuario)
Moto
Moto(String, int, boolean, boolean)
califica(boolean, int) int


Empresa
Empresa()
agregaPedido(Pedido) void
aceptaPedido(Pedido) void
muestraViajes(Cliente, LocalDate, LocalDate) void
sumaRecaudado(double) void
getViajes(Cliente) List<IViaje>
agregaChofer(Chofer) void
agregoViaje(IViaje) void
getViajes(Chofer) List<IViaje>
muestraViajes(Chofer, LocalDate, LocalDate) void
agregaVehiculo(Vehiculo) void
vehiculoConViaje Vehiculo
clientes List<Cliente>
vehiculosEnUso List<Vehiculo>
pedidos List<Pedido>
vehiculos List<Vehiculo>
choferesDisponibles List<Chofer>
viajes List<IViaje>
choferDisponible Chofer
recaudado double
usuarios List<Usuario>
vehiculosDisponibles List<Vehiculo>
choferConViaje Chofer
viajesFinalizados List<IViaje>
vehiculoDisponible Vehiculo
choferesEnUso List<Chofer>
choferes List<Chofer>
viajeFinalizado IViaje
instance Empresa
ViajeZonaPeligrosa
ViajeZonaPeligrosa(Pedido, Chofer, Vehiculo)
costo double
DecoratorEquipajeBaul
DecoratorEquipajeBaul(IViaje)
costo double
pedido Pedido
ViajeZonaEstandar
ViajeZonaEstandar(Pedido, Chofer, Vehiculo)
costo double
ViajeZonaCalleSinAfaltar
ViajeZonaCalleSinAfaltar(Pedido, Chofer, Vehiculo)
costo double
DecoratorConMascota
DecoratorConMascota(IViaje)
costo double
pedido Pedido

Flujo completo desde el ingreso de un pedido hasta su finalización

A continuación se detalla brevemente cual es el flujo esperado al ingresar un pedido a la empresa de transporte hasta el momento de finalizar dicho pedido.

Ingreso del pedido

Se ingresa un pedido a partir de un cliente, el método a utilizar es el siguiente:

```
public Pedido(String zona, boolean mascota, String equipaje, int cantPersonas, Cliente cliente,double distancia)
```

Anterior a esto se crea la empresa, los choferes, los vehículos y usuarios.

Una vez realizado el pedido se evalúa para verificar que se cumplan las condiciones

```
public void evaluaPedido(Pedido p)  {  2 usages
    List<Vehiculo> vehiculos=empresa.getVehiculos();
    Vehiculo v=null;
    try {
        buscarVehiculo(vehiculos,p.getCantPersonas(),p.usoBaul(),p.getMascota());
    } catch (NoVehiculoException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
    empresa.agregaPedido(p);
}
```

En caso de que no se encuentre un vehículo que satisfaga la solicitud del cliente se lanza una excepción comunicando su razón, pasando a estar rechazado el pedido.
También se puede llegar a lanzar una excepción en caso de que no se encuentre ningún chofer en la empresa.

En caso contrario, si se encuentra tanto un chofer como un vehículo que satisfaga dicha solicitud; el Pedido pasa a estar aceptado y se genera un Viaje en situación de “solicitado”.
Una vez que el cliente suba al vehículo la situación del Viaje pasa a ser “iniciado”.
Al finalizar dicho Viaje, se cambia la situación a “pagado”.
Una vez pagado, el chofer se encarga de actualizar la situación del Viaje como “finalizado”.

Paralelo a esto existen los modulos GestionPagoChoferes y GestionPuntajes en donde el administrador se encarga de actualizar a fin de mes el sueldo de los choferes y sus respectivos puntajes.

Patrones aplicados

En la totalidad del trabajo se utilizaron los siguientes patrones de diseño:

- Decorator
- Singleton
- Factory

El patrón Decorator desempeña un papel fundamental en la creación y personalización de los Viajes dentro del sistema. Su función principal radica en la capacidad de agregar dinámicamente características y funcionalidades adicionales a los Viajes, según lo requerido en el Pedido. Esta capacidad de añadir características nuevas de forma dinámica es especialmente valiosa para evitar la conocida explosión clases, ya que permite construir Viajes únicos y personalizados sin la necesidad de crear una gran cantidad de clases diferentes.

El patrón Singleton se utilizó para la creación de la clase Empresa, de esta forma nos aseguramos que exista una única instancia de esta a la vez que proporcionamos acceso global a ella. Es fundamental para que haya una gestión eficiente de choferes, vehículos, clientes y otros recursos de la empresa de transporte.

El patrón Factory fue utilizado para crear instancias de Usuarios, Viajes y Vehículos en nuestra aplicación. Esto es importante porque debemos considerar diferentes tipos de usuarios (administrador, chofer o cliente), diferentes tipos de viajes (zona estándar, peligrosa, sin asfaltar) y diferentes tipos de vehículos (Combi, Moto y Automóvil). Esto nos permite centralizar la creación en una fábrica especializada según los parámetros dados, lo que mejora la claridad y permite la extensión futura del sistema.

Observaciones y mejoras durante el desarrollo

Durante el desarrollo de la clase Vehículo y sus métodos relacionados, nos planteamos la posibilidad de hacer abstractos ciertos métodos como verificaPasajeros, verificaBaul y verificaMascota, delegando su implementación a las clases derivadas como Automóvil, Combi y Moto. Sin embargo, al revisar detenidamente el código, nos dimos cuenta de que sería más eficiente y limpio generalizar estos métodos en la clase Vehículo. Esta decisión nos permitió reducir la duplicación de código y mejorar la mantenibilidad del sistema a largo plazo.
Además, surgieron dudas sobre la implementación específica de ciertas características, como la aceptación de mascotas por parte de los vehículos. Inicialmente consideramos dejar que cada vehículo decidiera por sí mismo si aceptaba mascotas, por ejemplo, las motos podrían rechazarlas. Sin embargo, luego reflexionamos sobre la importancia de considerar esta característica como un atributo específico de cada vehículo. Esto nos brinda flexibilidad para futuras mejoras, como la adición de accesorios para mascotas en las motos. Al incorporar este criterio en nuestro factory mediante la inicialización del atributo petFriendly, aseguramos que el código seguirá funcionando sin problemas en caso de futuras actualizaciones.

Para el diseño de la clase abstracta Viaje, optamos por utilizar una interfaz IViaje que define los métodos y propiedades básicas que deben tener todos los tipos de viajes en nuestra aplicación. Luego, creamos tres clases concretas que extienden de Viaje: ViajeZonaPeligrosa, ViajeZonaEstandar y ViajeZonaCalleSinAfaltar, cada una con sus características y comportamientos específicos según el tipo de zona en la que se realiza el viaje.
Además, implementamos un patrón Decorator para añadir funcionalidades adicionales a los viajes. Creamos la clase DecoratorViajes como base abstracta para los decoradores y luego desarrollamos los decoradores DecoratorConMascota y DecoratorEquipajeBaul, que agregan la capacidad de llevar mascotas y equipaje de baúl respectivamente a cualquier tipo de viaje.
Durante el proceso de diseño, consideramos la necesidad de crear un decorador adicional para casos específicos, como cuando el viaje no incluye mascotas o cuando se trata exclusivamente de equipaje de mano. Sin embargo, llegamos a la conclusión de que estos casos no requerían un decorador separado, ya que no modifican el precio del viaje ni afectan significativamente su funcionalidad básica.
Esta decisión nos permitió mantener la estructura del código más simple y coherente, evitando la creación de decoradores adicionales que no aportaban un valor significativo al sistema. Además, al tener una estructura modular y flexible con el uso de la interfaz y los decoradores, podemos agregar nuevas funcionalidades o tipos de viaje en el futuro de manera sencilla y sin afectar el diseño existente.

- Se consideraron las siguientes excepciones:

    - **NoChoferException:** Lanzada cuando no se encuentra Chofer en la Empresa o lanzada cuando no existe un chofer disponible para realizar el viaje.

    - **NoVehiculoException:** Lanzada cuando no se encontró un vehículo que cumpla con los requisitos del Pedido del cliente.

    - **UsuarioExistenteException:** Lanzada cuando se quiera crear una cuenta con un nombre de usuario ya existente en el sistema.

    - **ViajeNoEncontradoException:** Lanzada cuando se quiera cobrar un viaje inexistente.

    - **UsuarioInexistenteException:** Lanzada cuando se quiere buscar a un usuario y el mismo no existe.