TP PROGRA III

Integrantes:

Marcos Lascar

Valentin Señorans

Agustin Cerdá

Manuel Gracia

INTRODUCCIÓN

A la hora de comenzar con el trabajo sabíamos que el foco central iba a estar puesto en poder aplicar los conceptos teóricos vistos hasta ahora en la materia en el proyecto de manera adecuada, como por ejemplo un correcto uso del polimorfismo que vimos que es un pilar fundamental en la programación orientada a objetos.

Esto nos condujo a releer los conceptos en los que teníamos dudas para poder llevarlos a cabo de una manera correcta y lo más prolija posible, ya que al usar diferentes tipos de patrones, se nos iba a dificultar el manejo de datos de las diferentes clases.

Es por eso que a continuación explicamos cómo fuimos desarrollando cada una de las etapas del proyecto.

PATRÓN FACTORY

Creación de Vehículos

Se utilizó una clase llamada “VehiculoFactory” con el fin de implementar el patrón de diseño factory para la creación de vehículos que serán asignados a distintos choferes con el objetivo de proporcionar un viaje. Esto hace una fabricación de vehículos con su distintos tipos más fluida. En el caso de que el tipo de vehículo fabricado sea inexistente, se lanzará una excepción “TipoDeVehiculoInexistenteException” que informará al usuario que el tipo de vehículo ingresado fue incorrecto.

```
package negocio;


public class VehiculoFactory {

    public Vehiculo getVehiculo(String tipo, String patente) throws TipoVehiculoInexistenteException {
        Vehiculo respuesta = null;
        if (tipo.equalsIgnoreCase("Moto"))
            respuesta = new Moto(patente);
        else
            if (tipo.equalsIgnoreCase("Auto"))
                respuesta = new Auto(patente);
            else
              if (tipo.equalsIgnoreCase("Combi"))
                  respuesta = new Combi(patente);
              else
                  throw new TipoVehiculoInexistenteException();
        return respuesta;
    }
}
```

Creación de Viajes

El patrón factory para los viajes es implementado de la misma manera que el anterior, pero agregando la funcionalidad de los Decorator (Patrón Decorator y como fue implementado, explicado en la siguiente página).

De manera que creamos un encapsulado de tipo IViaje, el cual lo instanciamos de primera mano con una Zona, proceso el cual si falla (fallará en caso de que el tipo de zona ingresada en el formulario de pedido no exista en nuestras opciones de zonas) lanzará la excepción **ZonaInexistenteException**, y si no, continuará aplicando Decorator Mascota y luego Decorator Baul.

CREACIÓN DE CHOFERES

Intentamos también aplicar el patrón factory para la creación de distintos tipos de choferes (permanente, temporario o contratado), pero nos dimos cuenta que los parámetros para la creación de cada tipo de chofer eran distintos, ya que los sueldos se calculaban de diferentes maneras. Entonces terminamos optando por la creación de distintos métodos para los diferentes tipos de choferes

```
public void agregarContratado(String dni, String nombre) {
    Chofer chofer = new Contratado(dni, nombre);
    choferes.add(chofer);
}

public void agregarTemporal(String dni, String nombre, double sueldo_basico, int cantViajes) {
    Chofer chofer = new Temporario(dni, nombre, sueldo_basico, cantViajes);
    choferes.add(chofer);
}

public void agregarPermanente(String dni, String nombre, double sueldo_basico, int cantHijos, GregorianCalendar fecha_ingreso) {
    Chofer chofer = new Permanente(dni, nombre, sueldo_basico, cantHijos, fecha_ingreso);
    choferes.add(chofer);
}
```

Al momento de *testear el programa* y todos los métodos de Empresa, nos encontramos con que algunos de ellos para ser llamados necesitábamos los Objetos **cliente** y/o **chofer** con lo cual, modificamos los métodos de la imagen con el objetivo de que devuelvan el objeto creado, para así utilizarlos en el main.

PATRÓN DECORATOR

Utilizamos el patrón decorator para los viajes, ya que éstos poseen diferentes aumentos de precios dependiendo su **Zona** destino, si lleva **Mascota** y si requiere **Baúl**, dichos aumentos son directamente proporcionales a la cantidad de pasajeros y la cantidad de kilómetros.
Donde las clases de las zonas *(ZonaPeligrosa, ZonaEstandar, CalleTierra)* extenderán de la clase abstracta *Viaje,* la cual será luego decorada por una extensión de *DecoratorMascota (DecoratorConMascota o DecoratorSinMascota)* y otra de *DecoratorBaul (DecoratorConBaul o DecoratorSinBaul).*
Todas las anteriores a su vez, implementan la interfaz *IViaje*, para así poder ser parte de un mismo grupo y tener métodos en común.
A su vez, dentro de estos fue utilizado el patrón template (explicado más adelante), al momento de calcular el costo, ya que **getCosto()** suele estar definido de manera concreta en una clase abstracta, conteniendo en su interior las clases abstractas **getIncKilometros()** y **getIncPasajeros()**, que luego son sobreescritas al momento de extender la clase abstracta padre.

PATRÓN SINGLETON Y FACADE

Utilizamos patrón singleton y facade en la clase Empresa. Es singleton ya que solo debemos poseer una **única instancia**, la cual es la que sufrirá todas las modificaciones. Y al ser facade, conseguimos que esta clase sea la que deberá procesar todo tipo de

solicitudes, ya sea de parte del **Administrador** o del **Cliente**, y será la que contenga la lista de Choferes, Vehículos y Clientes que componen al Servicio de transporte. Delegando las solicitudes a sus correspondientes clases o modificando las actuales.

PATRÓN TEMPLATE

Utilizamos el patrón template con tal de definir los pasos comunes de un algoritmo.
Uno de los contenidos en donde lo tomamos como consideración es en la clase abstracta Vehículo que se encarga de evaluar si las condiciones de pedido se cumplen para cada tipo de vehículo y si es el caso realiza el cálculo de la prioridad correspondiente. Donde en cada clase concreta se comportará de una manera diferente.
Primero considera si la cantidad de pasajeros declarada en el pedido es soportado por el tipo de vehículo correspondiente, luego contempla si el pedido cuenta con mascota y si el vehículo que corresponde puede contar con ella, mismo caso para baúl.
Una vez evaluados estos pasos se ejecuta un paso final que se encarga de realizar el cálculo de la prioridad, en caso contrario retornará un valor que representa que el pedido no puede ser suministrado por ese vehículo.

MANEJO DE EXCEPCIONES Y POST CONDICIONES

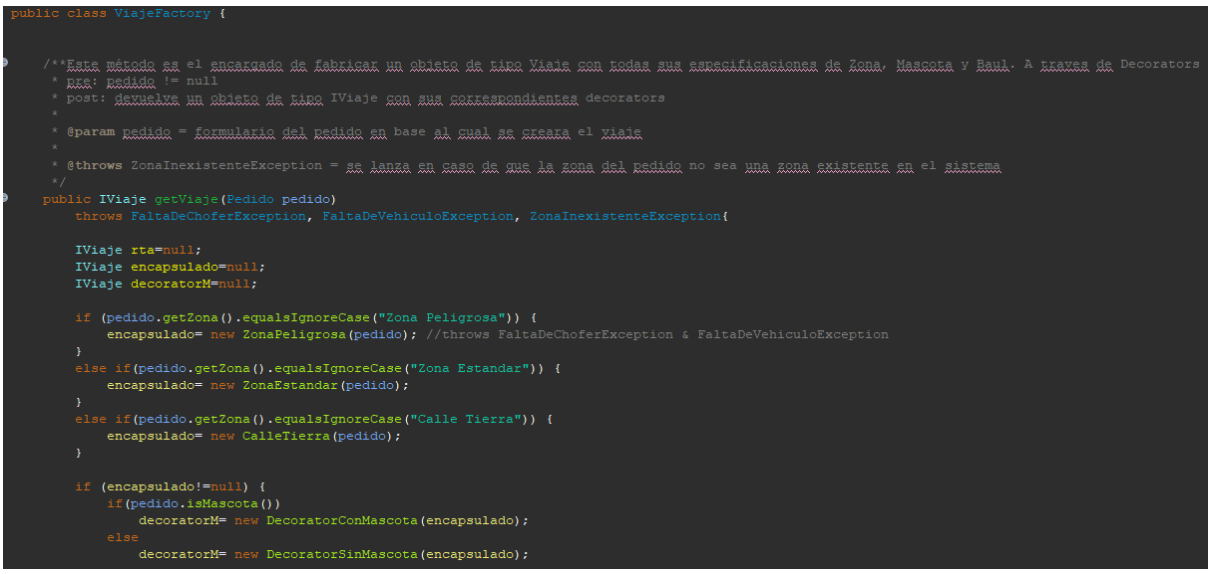
A lo largo del Trabajo Práctico implementamos diferentes estrategias para evitar o manejar errores, a través de:

- precondiciones: condición que debe cumplirse antes de ejecutar un método o código. Define el estado esperado del entorno o los datos antes de la acción. Su propósito es garantizar la validez y seguridad de la operación.

- post condiciones: condición que se espera que sea verdadera después de que se haya ejecutado un método o una sección de código. Indica el estado o resultado esperado después de completar una acción.

- excepciones: evento inesperado o no deseado que ocurre durante la ejecución de un programa y que interrumpe el flujo normal de ejecución. Las excepciones se utilizan para manejar situaciones de error o condiciones excepcionales.

Por ejemplo en ciertos métodos utilizamos solo precondiciones y postcondiciones, mientras que en otros tuvimos tuvimos que usar pre, postcondiciones y Excepciones, tal como es el caso del método público **getViaje()** de la clase ViajeFactory, mostrado a continuación:



```java
public class ViajeFactory {


    /**Este método es el encargado de fabricar un objeto de tipo Viaje con todas sus especificaciones de Zona, Mascota y Baul. A traves de Decorators
     * pre: pedido != null
     * post: devuelve un objeto de tipo IViaje con sus correspondientes decorators
     *
     * @param pedido = formulario del pedido en base al cual se creara el viaje
     *
     * @throws ZonaInexistenteException = se lanza en caso de que la zona del pedido no sea una zona existente en el sistema
     */
    public IViaje getViaje(Pedido pedido)
        throws FaltaDeChoferException, FaltaDeVehiculoException, ZonaInexistenteException{

        IViaje rta=null;
        IViaje encapsulado=null;
        IViaje decoratorM=null;

        if (pedido.getZona().equalsIgnoreCase("Zona Peligrosa")) {
            encapsulado= new ZonaPeligrosa(pedido); //throws FaltaDeChoferException & FaltaDeVehiculoException
        }
        else if(pedido.getZona().equalsIgnoreCase("Zona Estandar")) {
            encapsulado= new ZonaEstandar(pedido);
        }
        else if(pedido.getZona().equalsIgnoreCase("Calle Tierra")) {
            encapsulado= new CalleTierra(pedido);
        }

        if (encapsulado!=null) {
            if(pedido.isMascota())
                decoratorM= new DecoratorConMascota(encapsulado);
            else
                decoratorM= new DecoratorSinMascota(encapsulado);
```

Tal como se puede observar, devuelve un objeto de tipo IViaje, siempre y cuando no se lance la excepción `ZonaInexistenteException()`, que se ejecuta si la zona no existe en el sistema.

DECISIONES EN EL CAMINO

El proceso evolutivo del **viaje** está codificado ignorando el hecho de que los **choferes** se mantienen ocupados durante la duración del trayecto desde que éste pasa de Iniciado a Finalizado. Es decir, al momento de elegir un chofer, éste se va *instantáneamente* al fondo del ArrayList choferes. Aunque para esta primera parte no resultará un inconveniente, somos conscientes que lo vamos a tener que modificar para la segunda entrega. Esto mismo ocurre con los **vehículos** que son utilizados en cada viaje.

Para la **clonación** al momento de reordenar los viajes según su costo, no realizamos clonación profunda, incluso sabiendo que algunos de sus parámetros que son objetos tienen relación de composición con Viaje, porque no nos pareció necesaria para el uso que le vamos a dar a estas listas, ya que únicamente imprimimos por pantalla datos no referidos a los objetos que posee cada viaje. Vale aclarar que tuvimos que realizar una clonación por capas para calcular y mostrar correctamente el precio de cada viaje.

El parámetro **cliente** que recibe el constructor del pedido, aunque lo pusimos como *precondición*, nunca recibirá **cliente** como null en la segunda parte, ya que el que solicita el viaje en una primera instancia será un **cliente**, enviándose a sí mismo como parámetro. Esto no tomará forma hasta la correcta división de roles entre Administrador y Cliente

Nos ocurrió algo similar con el desarrollo de los métodos de **“CostoViajesMes”** y **“CantViajesMes”** donde recibirán como parámetro un objeto de tipo chofer contratado y temporario respectivamente. Si bien en las condiciones aclaramos que los objetos son distintos de NULL, esto no ocurrirá ya que se referencia en el método el tipo correspondiente de chofer como parámetro. También contemplamos que al invocar estos

métodos se tenga en cuenta el último mes presente al final de la lista, considerándolo como el mes actual para estas operaciones, con el fin de evitar un recorrido más extenso y optar por una lógica de suposición donde la empresa debería solicitar el cálculo del sueldo una vez que cierre todas las operaciones del mes. Así mismo con la misma lógica supusimos que los viajes deben estar en orden, ya que consideramos que la empresa reporta su viaje inmediatamente se realice y evitar que se reporten en orden distinto de fechas.

Al momento de mostrar un listado por pantalla, ya sea listado de clientes o de choferes, entre otros, optamos por crear una variable “texto”, en la cual vamos acumulando en cada renglón la información necesaria a mostrar por pantalla de cada tipo y/o método. Es decir que cada uno de estos métodos devuelven un String para luego ser impreso en la clase Prueba. Decidimos elegir dicha resolución ya que no es recomendable tener “system.out.println” en las clases internas.

No tuvimos en consideración el uso de **asertos**, ya que estos son verdaderamente útiles al momento de debuggear el programa, y para ese momento ya lo teníamos funcionando correctamente.

LOGROS PERSONALES

Como logros personales, podemos afirmar que aprendimos a usar librerías externas como el Gregorian Calendar, el cual nos despejó muchas dudas que teníamos de cómo manipular fechas y hacer operaciones con las mismas, que resultaron siendo muy útiles a la hora de desarrollar ciertos métodos, como por ejemplo los de cálculo de salarios de los choferes.

Tuvimos una dificultad con respecto al cálculo de sueldos del chofer temporario y contratado, ya que no teníamos claro cómo acceder a la información de los viajes realizados por esos tipos de choferes.
Después de un tiempo pudimos identificar que se tenía que recorrer la lista de viajes realizados para recaudar la información de los últimos viajes de estos tipos de choferes, en ese momento logramos comprender cómo entra en juego la relación que tiene la empresa como singleton con la clase de chofer temporario y contratado.
Donde para el contratado teníamos que tener en cuenta la información que tiene la empresa con respecto a la cantidad de viajes que realizó el chofer contratado en el mes actual. Y para el caso del chofer temporario la cantidad de dinero generado, contemplando todos los viajes realizados en ese mes.

Un gran reto que se nos presentó fue la *clonación por capas* de Viaje, ya que éste debía ir clonando capa por capa los Decorator **mascota** y **baúl** hasta llegar a su **zona**. Esto nos resultó un problema en un principio porque clonaba simplemente teniendo en cuenta la **zona** e ignoraba sus Decorator. Pudimos arreglarlo y el string que devuelve muestraViajes() es una correcta clonación y ordenación de los **Viajes**, teniendo en cuenta su *costo final*.

CONCLUSIÓN

Si hablamos de nuestra experiencia realizando el TP podemos decir que la principal duda era si ir haciéndolo todos juntos a la vez, o dividirnos las tareas y después unir todo.
La principal opción fue hacerlo todos juntos, pero al poco tiempo nos dimos cuenta de que se nos complicaba el hecho de coordinar los horarios para estar los cuatro juntos por lo cual recurrimos a dividir las partes e ir avanzando cada uno por su cuenta, pero haciendo un encuentro virtual cada pocos días para ir contando cada uno sus respectivos avances e ir hablando acerca de las dudas y cosas que no nos estaban saliendo.
Pero mientras más avanzábamos las partes divididas se fueron relacionando con otras y tuvimos que comunicarnos más a menudo para poder resolver.
Esta fue la forma que implementamos para realizar el TP y la verdad que terminó siendo bastante efectiva, pero eso no quita que descartemos la posibilidad de hacerlo todos juntos para la parte dos, ya que consideramos que también es una buena forma de realizar este trabajo incluso de forma más grupal aún.

Mientras desarrollabámos el trabajo nos dimos cuenta de la importancia de los conceptos adquiridos durante la cursada, como se relacionan con las entidades brindadas en la consigna y como se ven reflejados a la hora del desarrollo de un proyecto. Y cuán importante es el polimorfismo en la programación orientada a objetos, sobre todo con la relación de la clase Empresa con los distintos tipos de clases distintas.