Práctica 1. Uso de patrones de diseño creacionales y estructurales en OO

Integrantes del grupo:

- Carmen Chunyin Fernández Núñez
- Pablo García Guijosa
- Marta Xiaoyang Moraga Hernández
- Jesús Navarrete Caparrós

Índice

Los diagramas se encuentran en unas imágenes aparte.

practica1/DiagramaEJ1.jpg

practica1/DiagramaEJ3.jpg

practica1/DiagramaEJ4.jpg

La numeración es por el ejercicio que representan. El DiagramaEJ1 es un caso especial porque se utiliza tanto para el Ejercicio 1 y 2.

Ejercicio 1. Patrón Factoría Abstracta en Java

En este ejercicio, seguimos el guión proporcionado por el profesor a la hora de crear el diagrama e implementar las clases. Para el manejo de las bicicletas y carreras, implementamos funciones que retiraban bicicletas específicas dadas un porcentaje.

En la interfaz FactoriaCarreteraYBicicleta, implementamos dos métodos “crearCarrera()” y “crearBicicleta()” en vez de crear uno solo que crease los dos objetos, debido a que en java se debe preestablecer qué tipo de objeto se devuelve.

Debido al uso de hebras, la clase carrera debe implementar el método run() de la interfaz Runnable.

Ejercicio 2. Patrón Factoría Abstracta + Patrón Prototipo

Tomamos la implementación del ejercicio 1 hecha en Java y la traducimos a Python (uso de abstractmethod (pues las clases abstractas y las interfaces no existen por defecto en Python), eliminar tipos de las variables, etc).

Además, añadimos el patrón prototipo, que implementamos para poder clonar Bicicletas. Para realizarlo, utilizamos la función "deepcopy" del paquete "copy", que implementa lo que necesitamos. Esto, como su nombre indica, nos permitirá crear una copia profunda y no superficial.

Ejercicio 3. Patrón libre

En este ejercicio se nos ha dado la oportunidad de ser mas creativos, creando nuestro propio programa con el patrón y lenguaje que quisiéramos, y finalmente nos hemos decantado por realizar en java la unión de los patrones software: Builder y Composite.

En el ejercicio en cuestión implementamos el patrón builder en la creación de empleados, que pueden ser a tiempo completo o a medio tiempo, gracias a la clase director que utiliza a EmpleadoBuilder, que declara los métodos específicos de un empleado, para construir diferentes trabajadores basados en las especificaciones proporcionadas. Y por otra parte, utilizamos el patrón Composite en la clase Departamento, que se utiliza para representar tanto elementos individuales como colecciones de elementos de manera uniforme. En este caso, utilizaremos este patrón para representar la estructura jerárquica de los departamentos y subdepartamentos.

El objetivo del ejercicio propuesto es realizar un programa que permita crear, agregar y eliminar empleados, y a su vez, poder añadirlo a un departamento o subdepartamento, reflejando la estructura organizativa de una empresa de manera intuitiva y sencilla.

Ejercicio 4. El patrón filtros de intercepción en java.

En este ejercicio, para el diseño se utiliza el patrón Filtros de Intercepción tal y como se indica en el ejercicio, y el estilo Modelo-Vista-Controlador.

La interfaz muestra al usuario la información (revoluciones, distancia recorrida, etc), a través de los botones se le indica al Controlador las órdenes y éste se comunica con el Cliente, quién hace una petición al Gestor de filtros. Esta petición llega hasta la Cadena de Filtros, que ejecuta los filtros y recalculan las revoluciones; con esto ejecuta el Objetivo, que actualiza sus datos. Después de la comunicación con el cliente, el Controlador ordena a la Vista que se actualice, y como ya se ha ejecutado Objetivo, toma los datos nuevos del Modelo.

Para la implementación se utiliza el diagrama resultante de la fase de diseño y se respetan las restricciones del ejercicio. Uno de los puntos más importantes y que hubo que solucionar fue el cómo realizar la actualización de valores.

Para ello, se utiliza un ScheduledExecutorService en el controlador para poder programar que cada segundo se ejecute una petición al cliente y que se actualice la interfaz. Gracias a esto, la velocidad seguirá aumentando o reduciéndose mientras que el acelerador o freno estén activos. No sólo eso, si no que además, si ninguno está activo o se apaga el motor, se irá perdiendo una velocidad equivalente al rozamiento. Esto es así porque un coche apagado o que no esté siendo acelerado ni frenado; no para en seco, si no que será afectado por el rozamiento hasta que pierda toda la velocidad.

Comportamiento de nuestra solución:

- La máxima velocidad del motor es 5000 RPM. Aunque, la final será 4970 porque tiene que luchar contra el rozamiento.
- Apagar el motor reinicia el cuentakilómetros reciente y hace que no se añadan al cuentakilómetros total kilómetros.
- Si el motor se apaga cuando tiene una velocidad, no para en seco. Pierde velocidad por el rozamiento, ni frena ni acelera.
- Todas las otras que da el ejercicio.

Para **ejecutar** el .jar:

java -jar --enable-preview .\Ejercicio4.jar

- Al ejecutar se abre la interfaz y en la línea de comandos se puede ver las peticiones que se van haciendo. Hay una petición cada segundo, como se ha explicado al hablar de la actualización de valores.
- Para cerrar el programa hacerlo desde la línea de comandos con Ctrl + C.