Informes de los ejercicios 1 y 2 de la Práctica de Diseño.

Daniela Alejandra Cisneros Sande. Usuario: d.cisneross@udc.es. Grupo: 2.1.
Mauro Moran Marcos. Usuario: mauro.moranm@udc.es. Grupo: 2.1.

Ejercicio 1: Gestión de Billetes.

Hemos implementado diferentes Principios para la realización de este ejercicio, estos son:

Principio de Responsabilidad Única: la utilización de este principio la podemos observar en todas las clases creadas (Origen, Destino, Precio, Fecha, Búsqueda) ya que cada una de estas clases tiene una única responsabilidad propia, por ende tienen una sola razón de cambio; si dicha responsabilidad cambiará es necesario cambiar el código que implementa a la clase. Implementar más de una responsabilidad, implicaría que un cambio en una de ellas podría llegar a afectar a las demás conduciendonos a un diseño frágil (con excepción del caso que dichas responsabilidades estén relacionadas y suelen cambiar de manera conjunta).

Principio Abierto-Cerrado: lo aplicamos a lo largo del ejercicio ya que de la clase principal (Criterios) vamos extendiendo la implementación de la funcion hacerBusqueda() por medio de herencia (sobreescribiendo al mismo) en cada una de las clases, así es como conseguimos que las clases puedan cambiar lo que hacen sin tocar el código de la misma.

Principio de Inversión de la dependencia: nuestra solución contiene una clase Abstracta (Criterio) de la cual dependen todas las demás clases, lo mismo lo aplicamos a la función hacerBusqueda() la cual es abstracta y le damos la implementación especifica en cada una de las clases, asi cumpliriamos la aplicacion del principio antes nombrado.

Por otro lado, hemos utilizado un patrón resolviendo el ejercicio siguiendo el mismo:

Patrón Composición: llevamos a cabo este patrón en la clase Criterios, la cual juega el rol de Componente siendo hacerBusqueda() la operación que se delega a las subclases. Las clases Origen, Destino, Fecha, Precio juegan el rol de Componente concreto y por último Búsqueda será la clase que juegue el rol de Composición. Hemos decidido utilizar dicho patrón ya que en el enunciado se nos deja claro que el diseño debe ser flexible permitiendo añadir en un futuro nuevos componentes (criterios de búsqueda) sin tener que cambiar todo el código; además resulta un código más claro y sencillo.

Por último tenemos los dos tipos de diagrama que representan al ejercicio.

Diagrama de Clases:

Diagrama Dinámico:

Ejercicio 2: Planificador de tareas.

Hemos implementado diferentes Principios para la realización de este ejercicio, estos son:

Principio de responsabilidad única: el principio de responsabilidad única consiste en que cada objeto debe tener una única responsabilidad dentro de la funcionalidad total del código, y todos los servicios que ofrece una clase están estrechamente alineados con dicha responsabilidad. La aplicación de este principio es una aplicación general a todas las clases del código. La clase Tarea tiene la responsabilidad de albergar los atributos del objeto Tarea, ListaTareas es un objeto en el que se almacena como atributo una lista de tareas y que contiene dos métodos para modificar esta: addTarea para añadir una tarea a la lista y removeTarea para eliminar. La clase abstracta AlgoritmoOrdenacion es una clase que da una impmenentación por defecto a un método para ordenar la lista de tareas (esta clase también cumple el principio Abierto-Cerrado). Pasando a cada una de las clases que sobreescriben el método por defecto definido en AlgoritmoOrdenacion; tanto DependenciaFuerte, DependenciaDebil como OrdenJerarquico son clases que su responsabilidad es implementar dicho método y devolver la lista de tareas ordenada. Por último, la clase ComparadorNombre es una clase que implementa la interfaz Comparator del API de Java y cuya responsabilidad es hacer una comparación entre dos objetos Tarea basándose en uno de sus atributos: el nombre.

Principio Abierto-Cerrado: este principio permite que cualquier entidad software sea abierta para permitir su extensión, pero cerrada a la modificación. Esto quiere decir que el principio abierto-cerrado se cumple cuando una clase permite modificar lo que hace sin tocar el código de la propia clase, y esto se logra a través de la herencia. En este caso, la clase AlgoritmoOrdenación es una clase abstracta que implemente un método llamado ordena, que pasándole la lista de tareas a realizar devuelve una lista de tareas en el orden de realización. La implementación por defecto que se le ha dado a este método es el orden alfabético; y las clases DependenciaFuerte, DependenciaDébil y OrdenJerarquico lo que hacen es darle una nueva implementación al método ordena, sobreescribiendolo, sin necesidad de cambiar el código de la clase AlgoritmoOrdenacion.

Principio “encapsula lo que varía”: en este principio se basa el patrón utilizado para resolver el ejercicio (patrón estrategia), y el principio trata de intentar separar las partes del código que varían y separarlas de lo que permanece estable. Esto se ve reflejado en la clase abstracta AlgoritmoOrdenacion y las tres clases que sobreescriben el método definido en la primera: DependenciaFuerte, DependenciaDébil y OrdenJerarquico. Lo que varía es la manera de ordenar las tareas y el orden de su realización; por lo tanto se han creado tres clases que cada una implementa un algoritmo diferente.

Principio de inversión de la dependencia: consiste en depender de abstracciones en lugar de concreciones. Este principio se cumple debido a que se utilizan clases abstractas y interfaces para implementar determinados métodos de una forma que la implementación no sea visible para el cliente y además estén hechos para que se puedan modificar. La clase

ComparadorNombre implementa el método compare de la interfaz del API de Java Comparator; y AlgoritmoOrdenacion es una clase abstracta que tiene un método ordena al que le da una implementación por defecto y otras clases como DependenciaFuerte, DependenciaDebil y OrdenJerarquico sobreescriben este método dándole una implementación diferente.

Por otro lado, hemos utilizado un patrón resolviendo el ejercicio siguiendo el mismo:

Patrón estrategia: el patrón seleccionado para la resolución del Planificador de tareas es el patrón estrategia, ya que este consiste en definir una familia de algoritmos, encapsularlos y hacerlos intercambiables. El patrón consta de tres elementos: el Contexto, que delega en el objeto Estrategia el cálculo del algoritmo; la Estrategia que declara una interfaz común para todos los algoritmos soportados y por último la Estrategia Concreta, que implementa el algoritmo utilizando el interfaz definido en el objeto Estrategia. Hemos decidido resolver el ejercicio utilizando el patrón estrategia debido a que el problema que se plantea consiste en das diferentes maneras o formas de utilizar un método: el algoritmo de ordenación .En este ejercicio, la clase ListaTareas juega el rol de contexto, AlgoritmoOrdenacion es una clase abstracta y juega el rol de Estrategia y las clases DependenciaFuerte, DependenciaDebil y OrdenJerarquico son tres diferentes Estrategias Concretas; cada una de ellas implementa un algoritmo sobreescribiendo el método ordena, que tiene una implementación por defecto en la clase abstracta AlgoritmoOrdenacion.

Por último tenemos los dos tipos de diagrama que representan al ejercicio.

Diagrama de Clases:

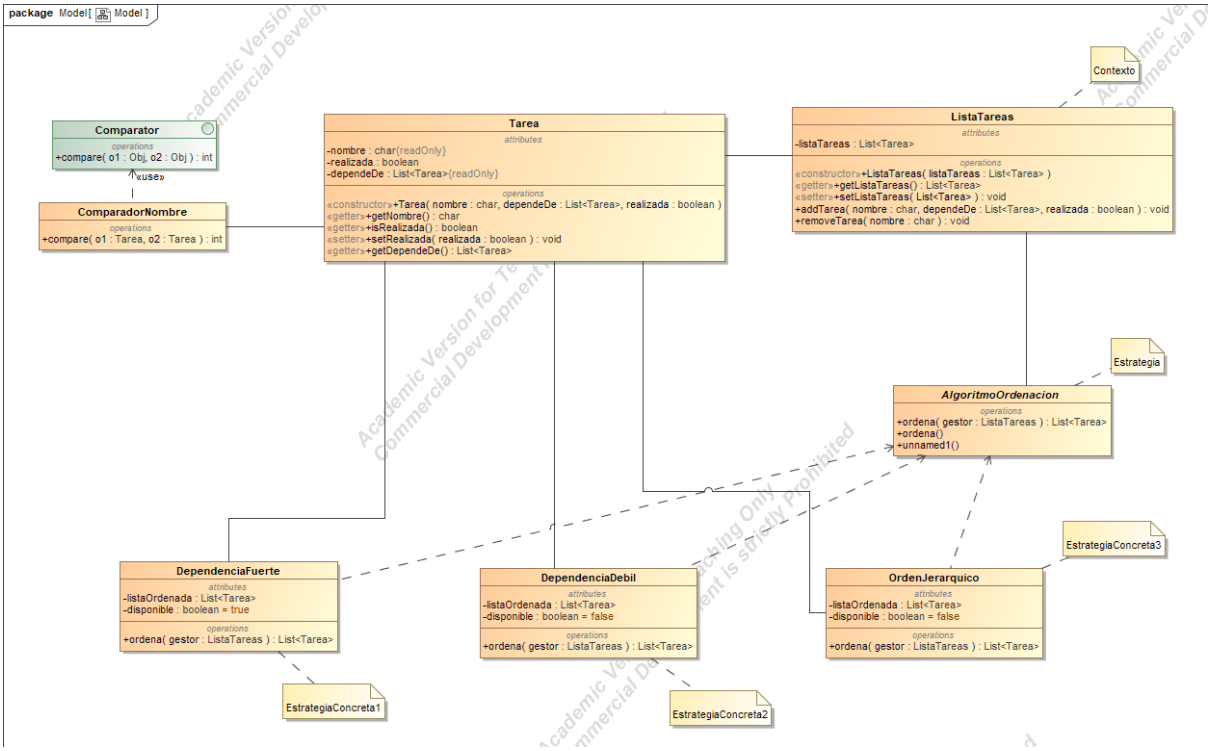

Diagrama Dinámico (diagrama de comunicación):