EJERCICIO 1 - GESTIÓN DE BILLETES

Principios de diseño

Se podría decir que en este ejercicio se respeta el principio de responsabilidad única ya que cada una de las clases tiene una responsabilidad única que está enteramente encapsulada en la clase.

De la misma manera, todos los servicios que provee el objeto están estrechamente alineados con dicha responsabilidad sin tener que depender de terceros ni sobrecargarse de métodos, limitando así la propagación de los cambios y facilitando el entendimiento des funciones de cada clase.

También se cree que se respeta el principio abierto-cerrado porque en la clase Builder anidada en Búsqueda, se definen los métodos para cada criterio.

De esta forma, se considera cerrada ya que cada criterio funciona independientemente del resto, entonces la clase está bien definida y de una forma estable.

Pero también se considera abierta ya que de una forma sencilla se podrían añadir nuevos métodos correspondientes a nuevos criterios sin modificar lo que ya había y ampliando el comportamiento de dicha clase, que es justo lo que se pide en el enunciado.

Patrones de diseño

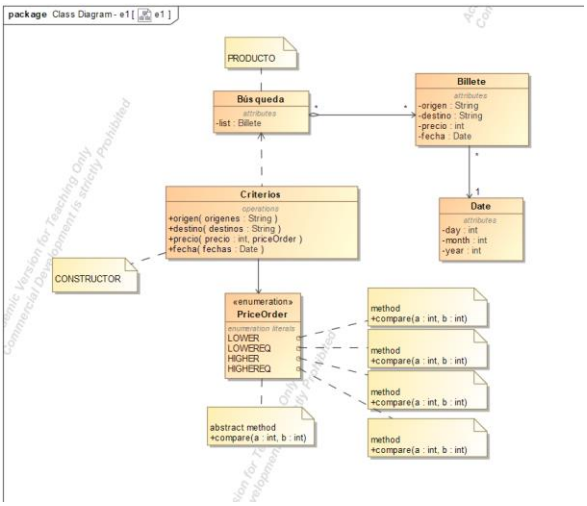
Patrón Builder

El patrón de diseño principal utilizado para darle una solución a este ejercicio ha sido el patrón Builder. El ejercicio consistía en una lista de billetes, la cual podía ser filtrada a partir de ciertos filtros, pudiendo estes mezclarse.

Para ello, decidimos crear una clase **Búsqueda**, con un atributo que es una lista de billetes. Dentro de esta clase implementamos una clase estática anidada llamada **Builder**, que va a ser la encargada de ir construyendo nuestro producto final, en nuestro caso la lista filtrada.

Dentro de la clase **Builder** estarán implementados métodos que se corresponderán a cada uno de los filtros, pudiendo elegir uno, varios o ninguno. También tendrá una lista de billetes como atributo que será la lista a la que le irán dando forma cada uno de los métodos de filtro. Al finalizar estos filtros se devolverá el objeto de la clase **Builder** ya construido.

Finalmente, tendríamos el objeto de la clase **Búsqueda** construido, y lo habría hecho internamente. La lista estaría filtrada con las opciones correspondientes y solo se tendría que acceder a ella a través del método *getList()* de esta clase.



En este ejercicio, es un caso especial en el que no se aplica un patrón de diseño común con un constructor abstracto que delega en constructores concretos. Aquí hay un único constructor que es la clase Criterios, que devuelve un producto de la clase Búsqueda, en este caso una lista de billetes, y el director sería cualquier clase que creara el objeto dado usando el constructor(Criterios).

Patrón Inmutable

Otro patrón que se ha tenido en cuenta es el patrón Inmutable. En este ejercicio, de la forma que nosotros decidimos resolverlo, se precisaban de clases individuales auxiliares. Estas son las clases **Billete** y **Date**.

Estas clases se crean para poder manipular billetes (con sus atributos *origen*, *destino*, *precio* y *fecha)* y poder manipular fechas (*día*, *mes* y *año*) con facilidad. Para cada objeto que se instancia en cada una de estas clases, interesa que sea el objeto definitivo y que no se pueda modificar.

Por eso, se utilizan las técnicas mencionadas en el patrón inmutable, como asegurarse de que la clase no puede ser extendida definiendo la clase como final, declarar todos los atributos como privados y finales, o evitar el acceso a componentes internos mutables inicializándolos con copias y no con referencias obtenidas por el cliente o devolviendo copias de seguridad cuando se accede a información de este objeto. Esto se puede ver en sus constructores y getters.

De la misma forma, hay una clase **Búsqueda** encargada de gestionar los servicios de filtrado de una lista de billetes. Esta clase también se implementa de forma inmutable, para que no pueda ser modificada y sea definitiva, y se siguen los mismos patrones que con las anteriormente mencionadas.

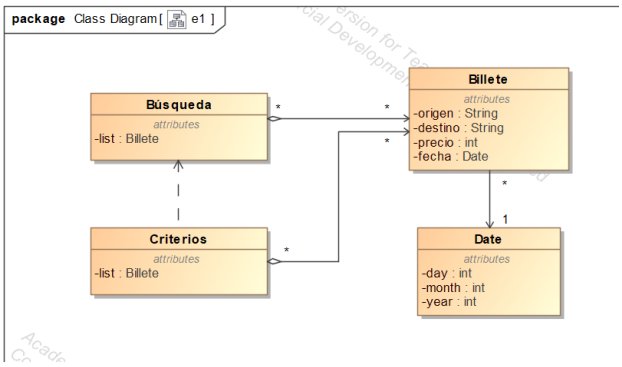
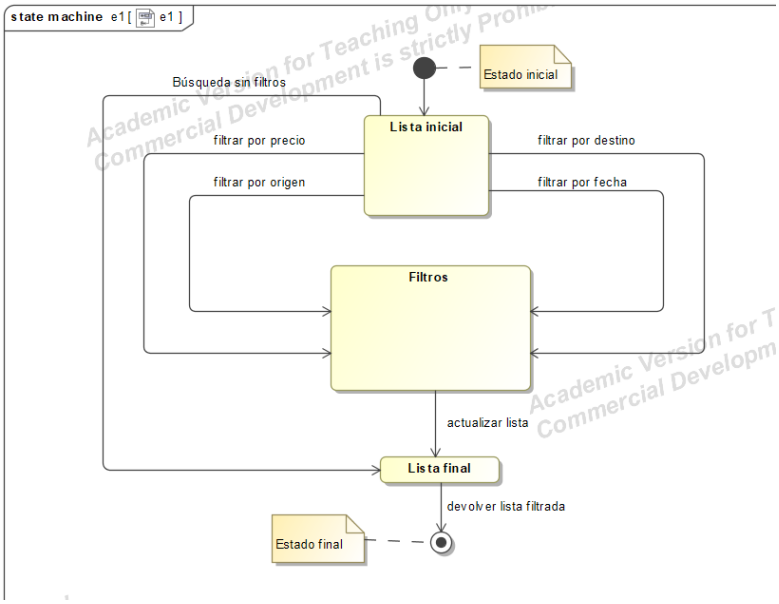


Diagrama dinámico - Diagrama de estados



EJERCICIO 2 – PLANIFICADOR DE TAREAS

Principios de Diseño

Para este ejercicio, los principales principios de diseño utilizados han sido el Principio de Responsabilidad Única y el de Inversión de la Dependencia.

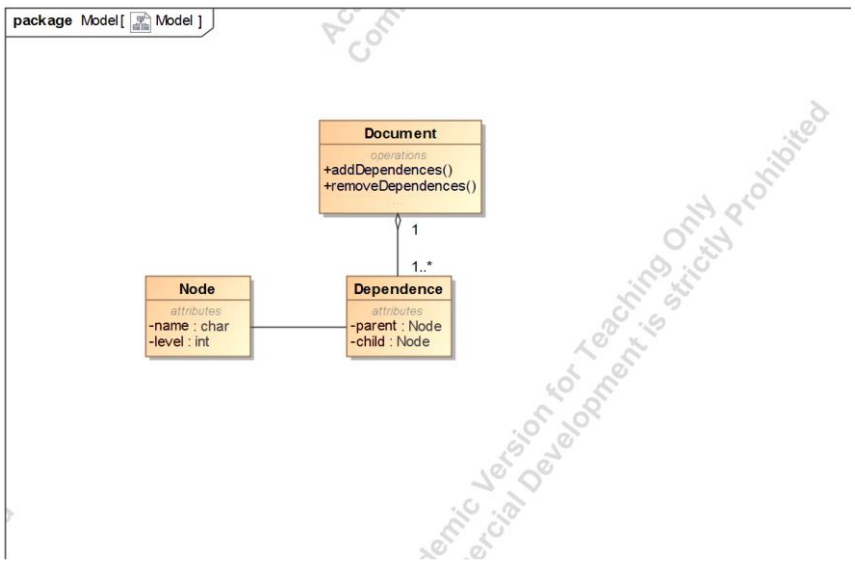
El Principio de Responsabilidad Única puede verse aplicado en la clase Graphic, que tiene la responsabilidad de gestionar el grafo; en la clase Document, la cual gestiona las dependencias de un proyecto de la empresa. La clase Dependence se encarga de asegurarse de que una dependencia tenga un nodo (tarea) padre y un hijo, y la clase Node se responsabiliza de que cada Nodo tenga un nombre tipo char y un nivel.

El Principio de Inversión de la Dependencia lo utilizamos en el atributo 'map' de la clase Graphic. Lo declaramos tipo Map pero utilizamos su implementación HashMap. También lo utilizamos en el método 'showLevel' de la misma clase para declarar una lista tipo List<Character> pero utilizar su implementación ArrayList, y lo mismo hacemos en el contstructor de la clase Document y su método getDocument(). En el caso de que nos interesara, sería muy sencillo cambiar a otra implementación de Map o List en estos casos.

Patrones de Diseño

Patrón Inmutable

Uno de los patrones de diseño utilizados es el Patrón Inmutable. Lo aplicamos para la clase Document, la cual tiene una serie de dependencias de tareas que no queremos que cambien, de la misma forma que no es muy cómodo eliminar nodos en un HashMap. De esta forma, para cada grafo que queramos hacer, necesitaremos un documento, y en el caso de querer modificar el grafo, habrá que crear un nuevo documento con las modificaciones oportunas y crear el grafo de nuevo.



Patrón Estrategia

El otro patrón de diseño utilizado es el Patrón Estrategia, para definir una familia de algoritmos para recorrer la lista de tareas de un proyecto con la interfaz TaskOrder, compuesta por tres clases para ordenar las tareas por dependencia fuerte, dependencia débil u orden jerárquico.

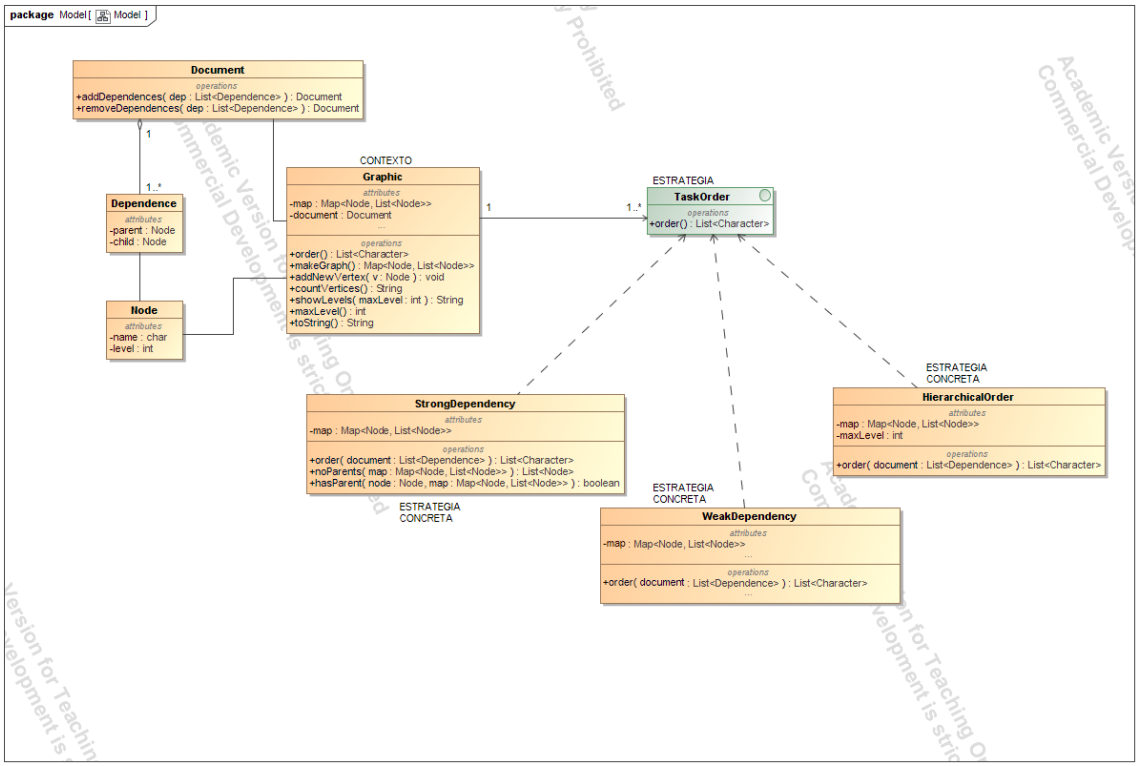


Diagrama dinámico – Diagrama de Secuencias