EJERCICIO NÚMERO 1

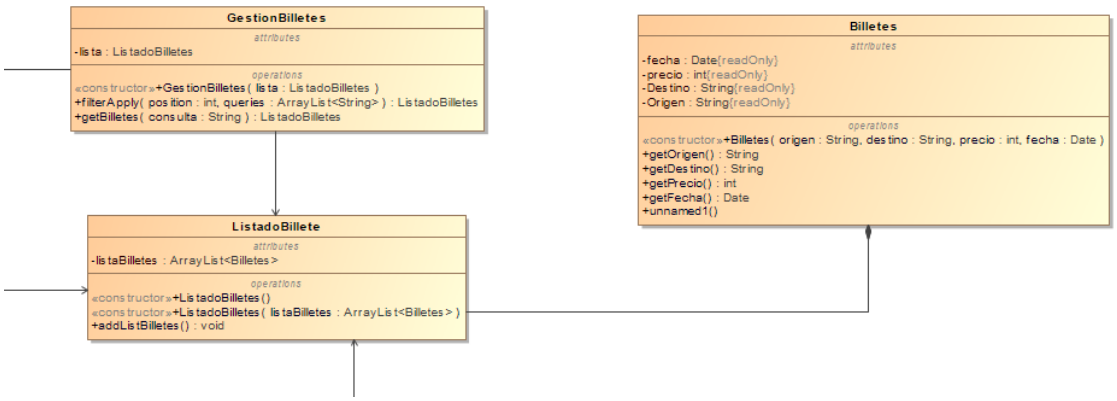
El problema por resolver nos presenta ante una creación de un software de gestión de expediciones de autobús, para fomentar el uso del transporte público. Reuniendo toda la información necesaria con los requisitos y problemáticas a solucionar empezaremos a trabajar.

La gestión a nivel superior del sistema de billetes de autobús funcionaría a partir de una cadena de texto como búsqueda. La consulta tiene que ser en el siguiente formato:

"destino=XX / origen=YY / precio (=,>,<,≥,≤) ZZ / fecha=dd/mm/yyyy"

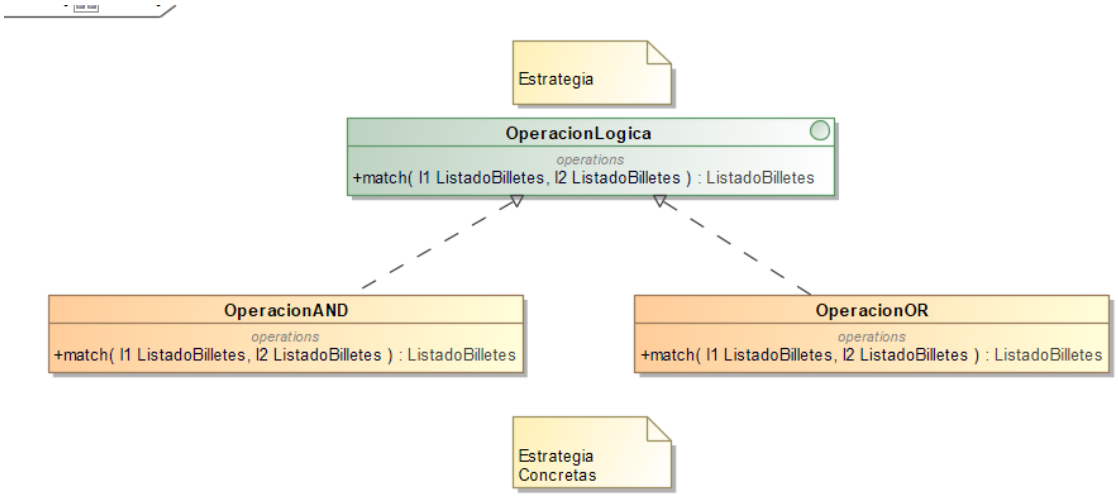
tendríamos que respetar también los espacios entre atributos (contando las operaciones lógicas)

Lo primero en que pensamos para realizar este ejercicio es en como representar tanto los billetes como el conjunto de billetes y la manera de gestionarlos a nivel de cliente de la plataforma. Para ello hicimos uso del **Patrón Composición**. Aunque no hagamos uso de interfaces, el esquema del patrón sería de la siguiente manera:



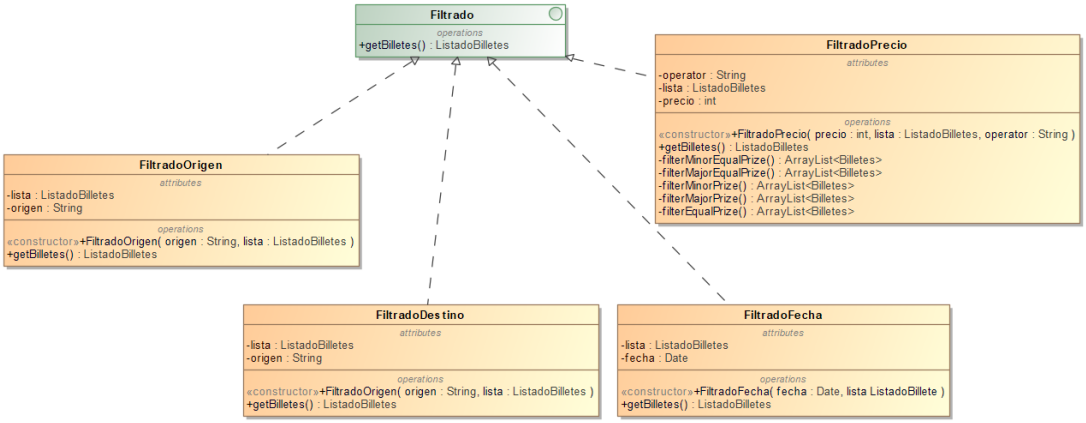
Así separamos la parte de creación y administración de billetes (en manos de la empresa de transportes) de la parte de búsquedas y consultas de los billetes disponibles a la venta (parte correspondiente al cliente de la plataforma). El listado de billetes no es nada más que una composición de billetes con sus métodos correspondientes

También en las consultas, pueden aparecer operaciones lógicas de tipo AND y OR, lo que añade un plus de complejidad al problema en el tratamiento de las listas. Para facilitar la implementación de estas dos operaciones, aplicamos el **Patrón Estrategia** cuyo esquema sería el siguiente:



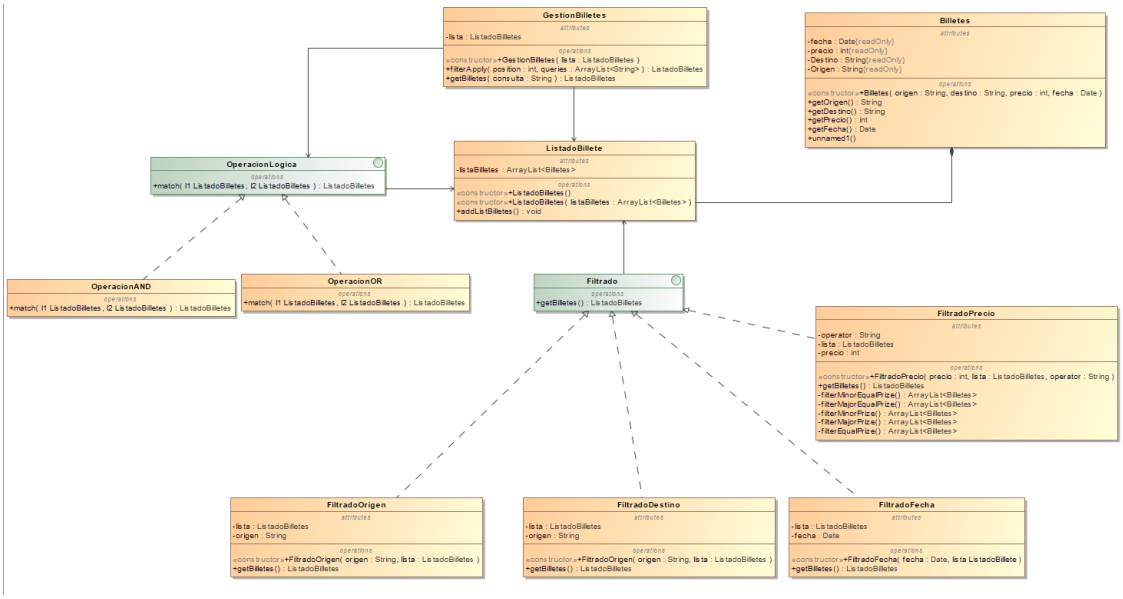
Hemos escogido este patrón porque nos ayuda a realizar como es debido cada una de las operaciones en el programa y adecuarlas así al resto del código. Una de las ventajas de este patrón es la fácil adaptabilidad a cambios en un futuro. Como pueden aparecer nuevas sentencias o a lo mejor se quiere utilizar otro tipo de operaciones lógicas, este patrón es el más adecuado para la implementación de esas operaciones.

Hablando de adaptabilidad a cambios, en este ejercicio nos encontramos con otra problemática que es el número de atributos de un billete. Como este número de atributos puede ir aumentando con el paso del tiempo, la utilización del **Patrón Estrategia** para aplicarlo en la búsqueda de billetes es la que escogimos. Creamos un filtro para cada atributo del billete, el cual utilizaremos dependiendo de lo que haya consultado el cliente en su búsqueda. El esquema sería el siguiente:



Con este diagrama podemos ver más claramente que, si en algún momento queremos añadirle algún atributo nuevo a los billetes (afectando así a los criterios de las consultas), simplemente crearemos un nuevo filtro correspondiente a este nuevo atributo y ahí crearemos sus criterios de búsqueda y haremos su correcto diseño.

El diagrama resultante total sería el siguiente: (subida imagen a git)



Por último tenemos que comentar los principios de diseño empleados apropiadamente en nuestro ejercicio. El principio de Responsabilidad Única está aplicado, por ejemplo, en las clases e interfaces de los Patrones Estrategia utilizados. En el momento que diferenciamos el filtrado de parámetros de búsqueda y los separamos a su vez de las operaciones lógicas AND y OR estamos aplicando este principio. También, al hacer por otra parte el procesado de la cadena de texto que introduce el cliente para buscar sus billetes, es decir; a los filtros de búsqueda les llega ya separada la cadena de texto y solo tienes que aplicar sus métodos sin tener que modificar la misma(se puede ver en la Clase GestionBilletes en el método filterApply).

Con la aplicación de estos patrones, podemos destacar también el uso del principio Abierto-Cerrado en el diseño. Nuestro código está en todo momento abierto a nuevas extensiones, ya sea añadiendo un nuevo filtro de búsqueda o añadiendo nuevas operaciones lógicas. Si queremos añadir un nuevo filtro, simplemente creamos una nueva clase que implemente a la interfaz de filtrado y esto no afectará a las clases ya hechas anteriormente, nunca las modificaría con un buen uso. También, al tratarse de interfaces, las clases se ven obligadas a implementar sus métodos a diferencia de las clases abstractas, hecho significativo que mantiene la premisa de “Cerrado a cambios”. Todo esto se puede apreciar observando las interfaces y sus clases implementadoras.