PRÁCTICA DISEÑO
EJERCICIO 1
GRUPO 23-03

Diego Suárez Ramos
Ander Varela

Introducción:

El gobierno quiere fomentar el uso del autobús metropolitano para desplazarse entre las distintas ciudades. Para ello, nos ha encargado un software de gestión de la expedición de billetes. La idea es que, los usuarios puedan buscar un billete por distintos criterios o una combinación de los mismos y luego, vistas las posibilidades, nos decidamos por el billete que más nos encaje en nuestras preferencias.

Implementación:

Tenemos una clase "Billetes" que guarda toda la información necesaria sobre un billete en concreto, luego tenemos la clase "Gestión" que almacena una lista de Billetes que son todos los billetes disponibles, y nos permite añadir billetes y filtrarlos según los diferentes criterios de búsqueda. Y por último, tenemos una interfaz genérica llamada "Criterios" que será la interfaz que tienen que implementar todos los Criterios/Operaciones de búsqueda y filtrado de billetes. De momento tenemos: Fecha, Destino, Origen, Precio como criterios y OR y AND como operaciones.

PRINCIPIOS USADOS:

-Responsabilidad única:

Cada objeto debe tener una responsabilidad única que esté enteramente encapsulada en la clase. Todos los servicios que provee el objeto están estrechamente alineados con dicha responsabilidad.

Intentamos hacer un diseño donde cada clase realiza únicamente una tarea. De esta manera tenemos que la clase "Billete" se encarga de manejar los billetes y guardar su información básica (origen, destino, precio, fecha); la clase "Gestión" se encarga de manejar todo lo que tiene que ver con la lista de billetes; y las diferentes clases que implementan la interfaz "Criterios" se encargan solo de filtrar los billetes.

-Principio de abierto-cerrado:

Las entidades software (clases, módulos, etc.) deberían ser abiertas para permitir su extensión, pero cerradas frente a la modificación.

Nuestro sistema es extensible porque permite agregar nuevos billetes a la lista de billetes disponibles y porque podemos crear nuevos criterios y operaciones de filtrado para los billetes, simplemente haciendo que implementen a "Criterios". Y además es cerrado, porque en ningún momento se muestra la implementación de las clases, ya que todo está definido como privado y no se puede modificar ningún atributo de ninguna clase, todos están declarados como final, por lo que está cerrada la modificación. Además, en Gestión se usa como interfaz externa la interfaz "List" de la API de Java (visible para el cliente), pero no es visible para un cliente, la clase que se usa de manera interna, que este caso es un "ArrayList".

-Principio de sustitución de Liskov:

Las clases derivadas deben ser sustituibles por sus clases base. Aquellos métodos que utilicen referencias a clases base deben ser capaces de usar objetos de clases derivadas sin saberlo.

Nuestra interfaz "Criterios" puede ser sustituida por cualquier clase que la implemente, debido a que no incluimos ninguna restricción en ningún método y además el funcionamiento interno de las clases es muy parecido. Por ejemplo, cuando usamos el método de filtrar, de la clase "Gestion", lo que se pasa como parámetro es un "Criterios", y se usa de manera externa la interfaz genérica de este, sin saber la implementación concreta que tiene, por lo tanto puedes pasar cualquier clase que implemente a "Criterios", que el método será capaz de usarlo sin saber que de que clase derivada se trata.

-Principio de inversión de la dependencia:

Depende de abstracciones y no de concreciones.

Cuando usamos el método de filtrar, de la clase "Gestion", en vez de depender de una clase concreta para filtrar, va depender de la interfaz "Criterios". De esta forma, tanto nosotros como cualquier otra persona que quiera añadir nuevos tipos de criterios o de operaciones podrá hacerlo sin comprometer el funcionamiento del sistema y sin tener que modificar nada de la clase "Gestion".

Aparte de los principios SOLID aplicamos los siguientes principios:

- Principio "Favorece la inmutabilidad":

Las clases deben ser inmutables, a no ser que haya una buena razón para hacerlas mutables.

Todas las clases son inmutables, debido a que no existen métodos que modifiquen el estado del objeto, nos aseguramos que todas las clases no puedan ser extendidas, definiéndolas todas como final(a excepción de la interfaz "Criterios" claro está, por el hecho de ser interfaz), todos los atributos de las clases están definidos como final y privados, y se evita el acceso a los componentes internos mutables.

-Principio encapsula lo que varía:

Identifica los aspectos de tu aplicación que varían y sepáralos de aquellos que permanecen estables

Básicamente sabemos que nuestro sistema va a permanecer todo estable, exceptuando las partes que dan lugar a cambio. Ya que, tanto los billetes como la gestión, siguen un diseño bastante estándar y que no suele variar con el paso del tiempo. Mientras que la parte que sí puede cambiar, son los distintos criterios que se usan para filtrar (los que implementan a "Criterios") la lista de billetes, ya que siempre podemos añadir más.

PATRONES:

-Patrón Composición:

Es un patrón estructural que se utiliza para componer objetos en estructuras de árbol que representan jerarquías todo-parte.

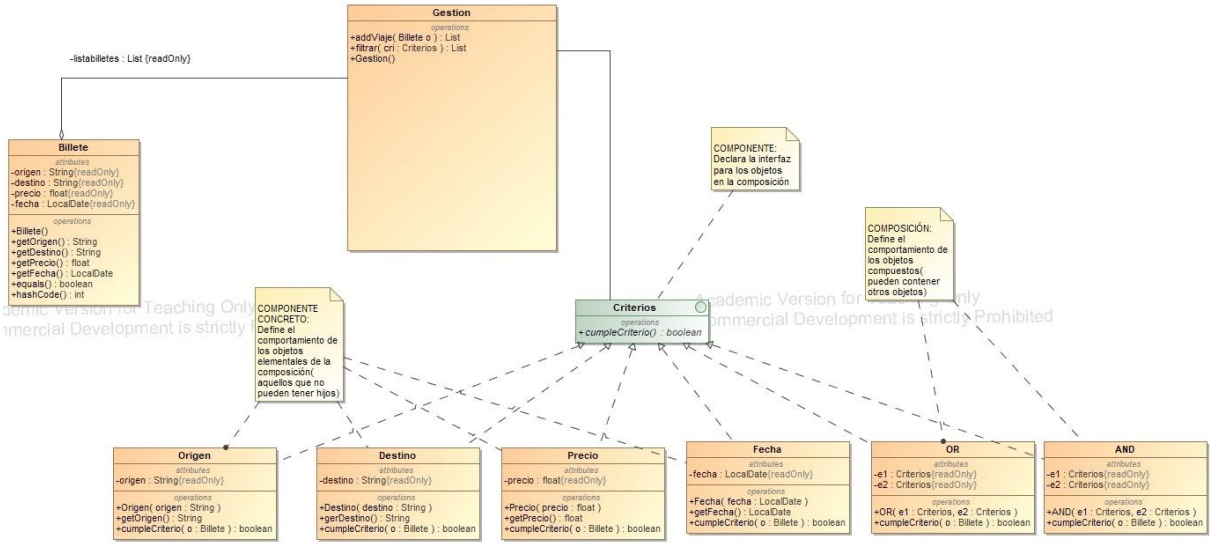
Este patrón se adapta perfectamente a lo que hacemos con los criterios, ya que si se mira atentamente nos damos cuenta de que se desarrolla una estructura en forma de árbol y que los objetos desarrollan los siguientes roles del patrón composición:

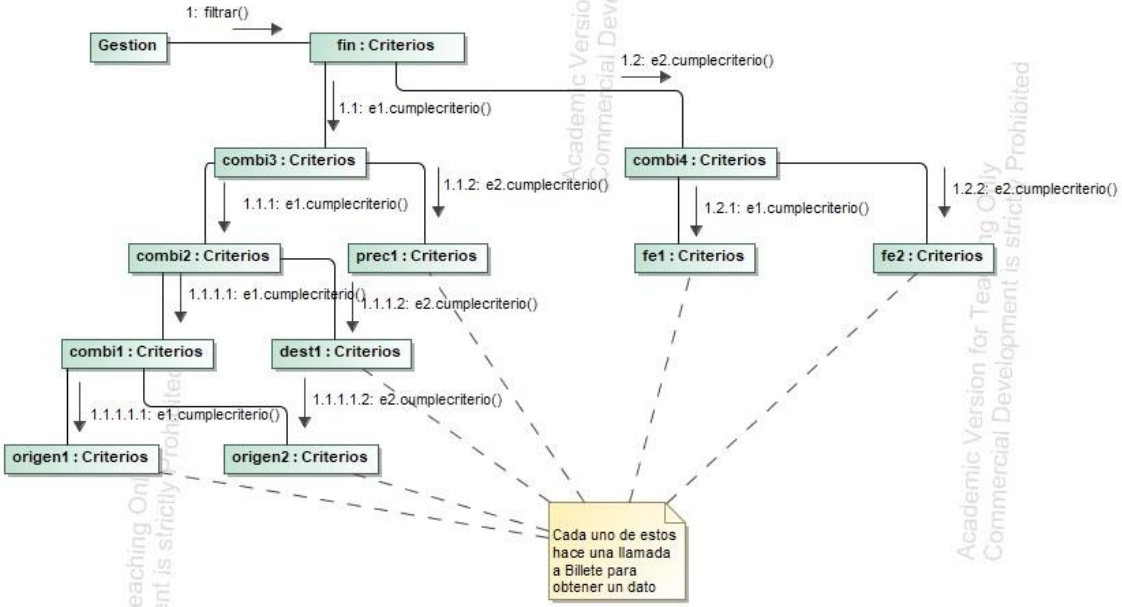
Componente: Es la interfaz "Criterios" la que desarrolla el rol de componente, ya que es la interfaz que implementan todos los objetos de la composición y que hace que implementen el método "cumplecriterio".

ComponenteConcreto: Las clases "Origen", "Destino", "Fecha" y "Precio" son las que desarrollan este rol. Son los objetos elementales de la composición, lo que significa que no pueden contener otros objetos. Cada una de ellas tiene diferentes implementaciones del método "cumplecriterio".

Composición: Las clases AND y OR son las que desarrollan este rol. Son los objetos compuestos, que pueden contener otros objetos que implementen a "Criterios". Además de tener diferentes implementaciones del método "cumplecriterio", también almacenan 2 "Criterios"

A continuación se muestra el diagrama de clases y el diagrama dinámico que en concreto es uno de comunicación:




1: filtrar()
Gestion
fin : Criterios
1.2: e2.cumplecriterio()
1.1: e1.cumplecriterio()
combi3 : Criterios
combi4 : Criterios
1.1.2: e2.cumplecriterio()
1.2.2: e2.cumplecriterio()
1.1.1: e1.cumplecriterio()
1.2.1: e1.cumplecriterio()
combi2 : Criterios
prec1 : Criterios
fe1 : Criterios
fe2 : Criterios
1.1.1.1: e1.cumplecriterio()
1.1.1.2: e2.cumplecriterio()
combi1 : Criterios
dest1 : Criterios
1.1.1.1.2: e2.cumplecriterio()
1.1.1.1.1: e1.cumplecriterio()
origen1 : Criterios
origen2 : Criterios
Cada uno de estos hace una llamada a Billete para obtener un dato