ALEX ANTELO MOURO / MATEO GALLO RODRÍGUEZ

GRUPO DE PRÁCTICAS : 21-02-PD

INTRODUCCIÓN

Vamos a explicar los patrones de diseño usados en los ejercicios 1 y 2, mostrando los diagramas UML correspondientes y los principios que cumplen.

EJERCICIO 1

PATRONES

Este ejercicio está programado en base a dos patrones: **el patrón composición** (el patrón principal), que se utiliza para componer objetos en estructuras de árbol que representan jerarquías todo-parte.

Primero declaramos una interfaz ("ComparadorBilletes"), que declara la operación "filtrar (List<Tickets>)", cuya definición será realizada por las clases que implementen dicha interfaz. Hay dos tipos de definiciones :

1 - **Componentes concretos**: definen los comportamientos elementales de la composición (clases "Precio"," Fecha", "Origen", "Destino").

2 - **Componentes compuestos:** definen los comportamientos compuestos de la composición (clases "Or" y "And").

Por otro lado, también hace uso del **patrón inmutable** para la definición de la clase Ticket.

Al utilizar estos patrones, estamos aplicando indirectamente los principios "**Favorece la inmutabilidad"** y **"Encapsula lo que varía"**.

PRINCIPIOS

Principios SOLID:

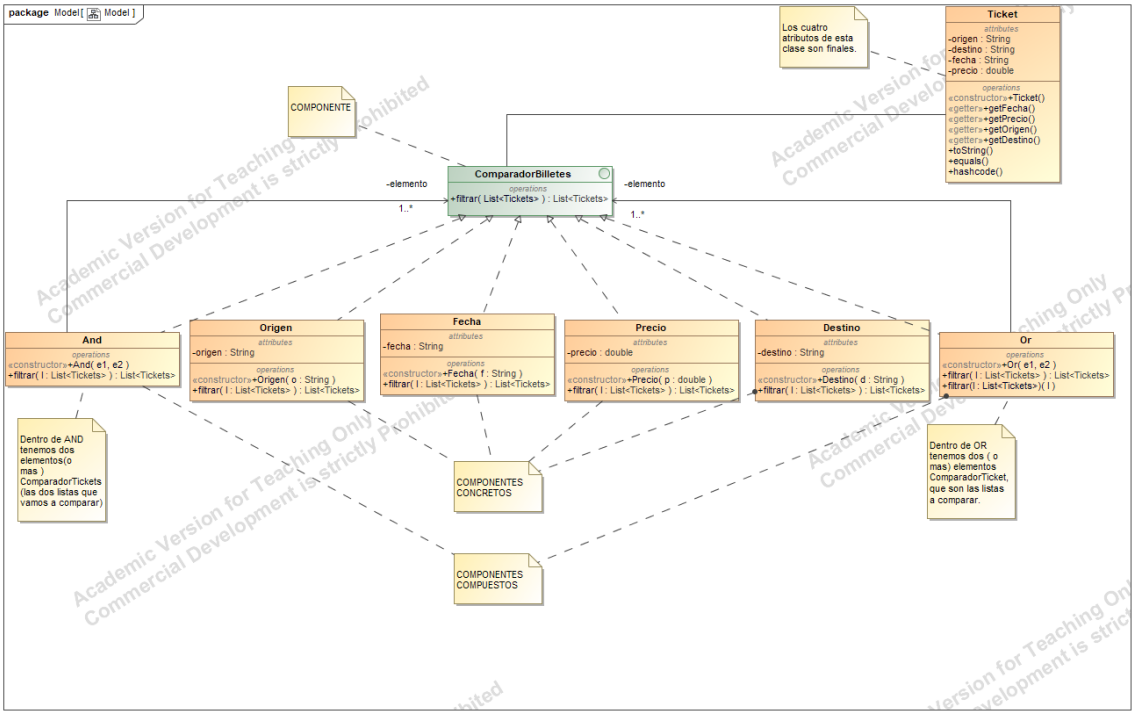
Responsabilidad Única: cada clase cumple la función que está destinada a realizar. En este caso, tanto los componentes concretos como los componentes compuestos cumplen un rol definido, las componentes concretas definen su propia función "filtrar (List<Character>)", y las componentes compuestas definen las funciones de And y Or.

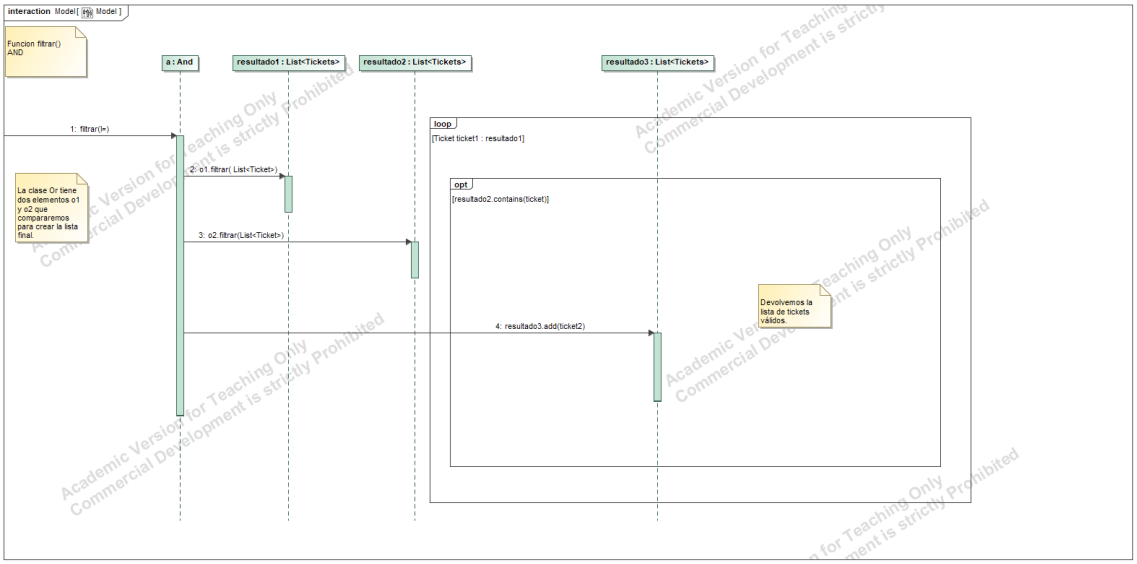
Abierto-Cerrado: las clases están abiertas para su uso, pero cerradas frente a su modificación, en este caso todas las clases vuelven a cumplir este principio ya que no se permite su alteración mediante un agente externo.

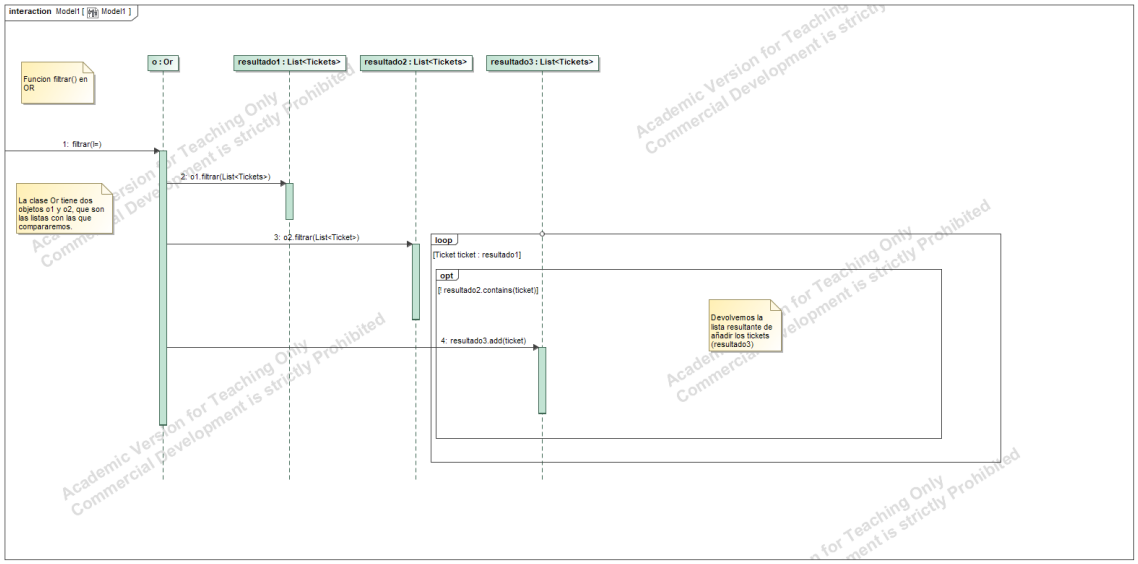
Sustitución de Liskov: por ejemplo, las clases And y Or reciben dos objetos sin saber que conducta particular del árbol deben seguir, pero no tiene problemas para ejecutar el método filtrar.

Inversión de la dependencia: a la hora de utilizar listas hacemos uso de una interfaz “List” (por ejemplo, en la función “filtrar(List<Tickets>)”) para así depender de la abstracción y no de implementaciones concretas como ArrayList o LinkedList.

Segregación de interfaz: Como solo tenemos una operación en la interfaz , este principio no es aplicable.







EJERCICIO 2

PATRONES

Este ejercicio está programado en base al **patrón estrategia**, que se divide en:

1 – **Contexto (“**Gráfico”**)**: delega en el objeto estrategia el cálculo del algoritmo.

2 – **Estrategia (“**OrdenTareas”**):** declara una interfaz común para todos los algoritmos.

3 – **Estrategia concreta (“**DependenciaDébil”, “DependenciaFuerte”, “OrdenJerárquico”**):**

implementa el algoritmo utilizando el interfaz definido en la Estrategia.

PRINCIPIOS

La explicación de los principios se ha hecho en el ejercicio anterior, por lo que solo vamos a mostrar ejemplos de los principios en este ejercicio.

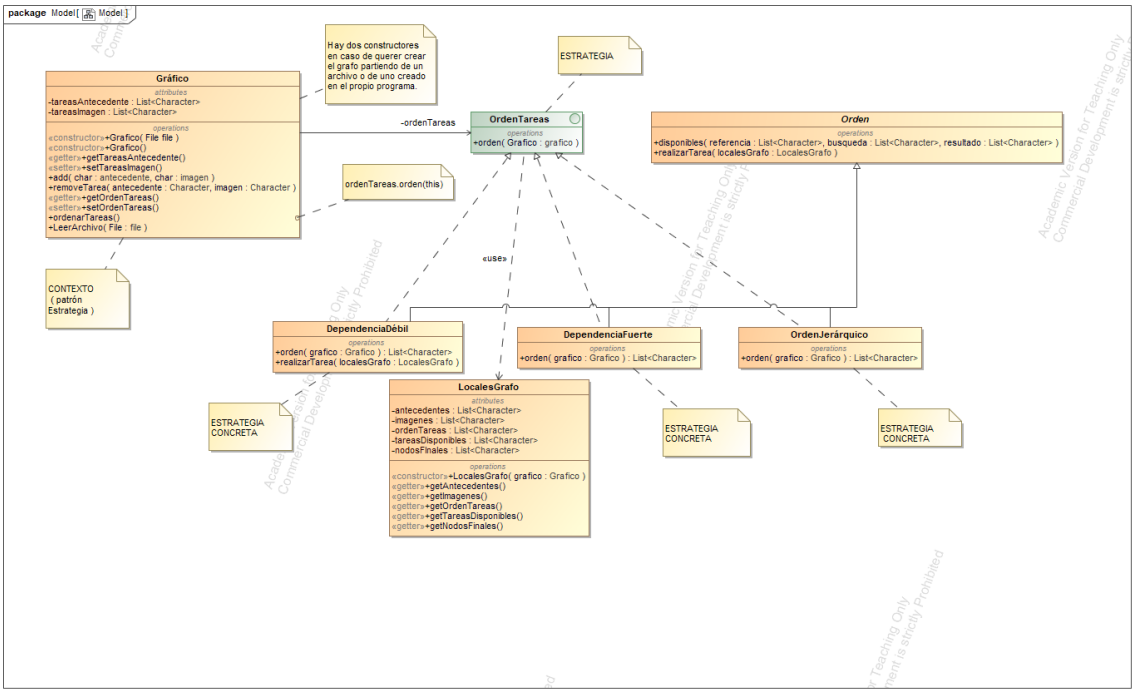
Responsabilidad Única: cada clase implementa su método orden (Grafico gráfico) que es su única función.

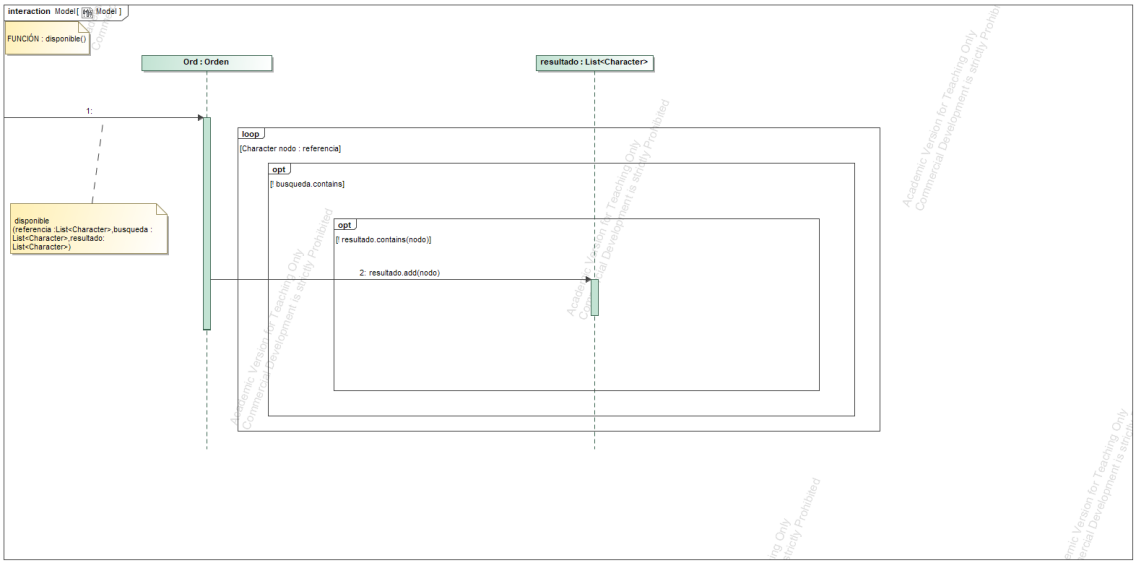
Abierto-Cerrado: un buen ejemplo es la clase LocalesGrafo.

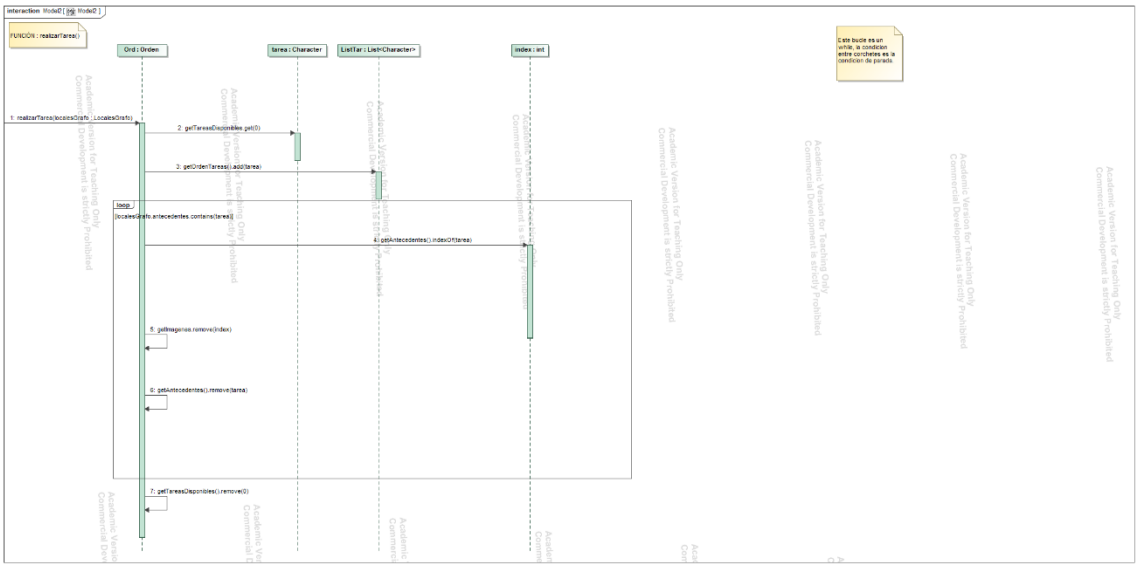
Sustitución de Liskov: en este caso en la clase Grafico tiene un atributo OrdenTareas cuya implementación desconocemos, pero que puede contener cualquier objeto que implemente el interfaz.

Inversión de la dependencia: de nuevo usamos listas de caracteres (List<Character>).

Segregación de interfaz: Como solo tenemos una operación en la interfaz , este principio no es aplicable.


package Model [Model]
Hay dos constructores en caso de querer crear el grafo partiendo de un archivo o de uno creado en el propio programa.
ESTRATEGIA
Gráfico
attributes
-tareasAntecedente : List<Character>
-tareasImagen : List<Character>
operations
«constructor»+Grafico(File file)
«constructor»+Grafico()
«getter»+getTareasAntecedente()
«setter»+setTareasImagen()
+add(char : antecedente, char : imagen)
+removeTarea(antecedente : Character, imagen : Character)
«getter»+getOrdenTareas()
«setter»+setOrdenTareas()
+ordenarTareas()
+LeerArchivo(File : file)
-ordenTareas
OrdenTareas
operations
+orden(Grafico : grafico)
Orden
operations
+disponibles(referencia : List<Character>, busqueda : List<Character>, resultado : List<Character>)
+realizarTarea(localesGrafo : LocalesGrafo)
ordenTareas.orden(this)
«use»
CONTEXTO
(patrón
Estrategia)
DependenciaDébil
operations
+orden(grafico : Grafico) : List<Character>
+realizarTarea(localesGrafo : LocalesGrafo)
DependenciaFuerte
operations
+orden(grafico : Grafico) : List<Character>
OrdenJerárquico
operations
+orden(grafico : Grafico) : List<Character>
LocalesGrafo
attributes
-antecedentes : List<Character>
-imagenes : List<Character>
-ordenTareas : List<Character>
-tareasDisponibles : List<Character>
-nodosFinales : List<Character>
operations
«constructor»+LocalesGrafo(grafico : Grafico)
«getter»+getAntecedentes()
«getter»+getImagenes()
«getter»+getOrdenTareas()
«getter»+getTareasDisponibles()
«getter»+getNodosFinales()
ESTRATEGIA CONCRETA
ESTRATEGIA CONCRETA
ESTRATEGIA CONCRETA


interaction Model [Model]
FUNCIÓN : disponible()
Ord : Orden
resultado : List<Character>
1:
loop
[Character nodo : referencia]
opt
[! busqueda.contains]
opt
[! resultado.contains(nodo)]
2: resultado.add(nodo)
disponible (referencia :List<Character>,busqueda : List<Character>,resultado: List<Character>)


interaction Model[ ] sd Model2 ( )
FUNCION : realizarTarea ()
Ord : Orden
tarea : Character
ListTar : List<Character>
index : int
Este bucle es un while, la condicion entre corchetes es la condicion de parada.
1: realizarTarea(localesGrafo : LocalesGrafo)
2: getTareasDisponibles.get(0)
3: getOrdenTareas().add(tarea)
loop
[localesGrafo.antecedentes.contains(tarea)]
4: getAntecedentes().indexOf(tarea)
5: getImagenes.remove(index)
6: getAntecedentes().remove(tarea)
7: getTareasDisponibles().remove(0)