Documentación Práctica de Diseño

Ejercicio 1.

Principios utilizados:

- Responsabilidad única:
  Consiste en que cada objeto tenga una única responsabilidad que esté enteramente encapsulada en la clase. En nuestro caso, lo utilizamos en las clases BuscarPorOrigen, BuscarPorDestino, BuscarPorPrecio, BuscarPorFecha y BuscarPorCriterios, puesto que su única función es buscar billetes según el criterio correspondiente y por tanto el cambio en una de ellas no afecta al resto.

- Abierto-cerrado:
  Este principio se basa en que al hacer cambiar lo que hace una clase no se afecte a su código. De esta forma, de cara al futuro, será muy fácil introducir cualquier tipo de cambio. En este caso, se consigue con el uso de la interfaz Buscar, puesto que, si posteriormente quisiéramos añadir un nuevo criterio, únicamente habría que introducir una nueva implementación de la interfaz Buscar. Por ejemplo si queremos añadir el criterio de búsqueda por provincia, habría que crear una nueva clase BuscarPorProvincia que implemente la interfaz Buscar.

- Inversión de la dependencia:
  Consiste en que los módulos de alto nivel no dependan de los módulos inferiores sino de abstracciones. Utilizamos este principio en el momento en el que declaramos una variable como Buscar y podemos utilizar cualquiera de las implementaciones (BuscarPorFecha, BuscarPorOrigen…). Un ejemplo sería: Buscar buscar = new BuscarPorPrecio(1.2f).

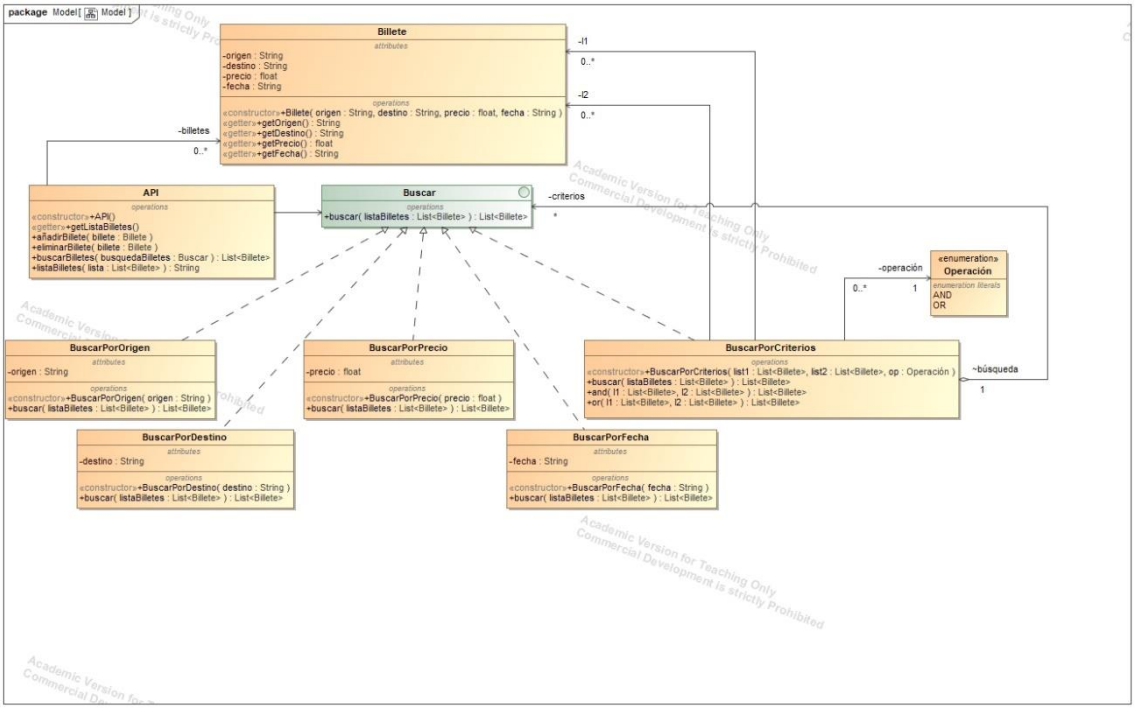
- Encapsula lo que varía:
  Se basa en identificar los aspectos de la aplicación que varían y separarlos de aquellos que permanecen estables. Lo utilizamos al separar las distintas formas de buscar los billetes.

- Bajo acoplamiento:
  Se centra en buscar diseños débilmente acoplados entre objetos que interactúan. Lo empleamos al crear objetos cohesivos, centrados en resolver una única responsabilidad (BuscarPorFecha, BuscarPorOrigen…), con bajo acoplamiento.

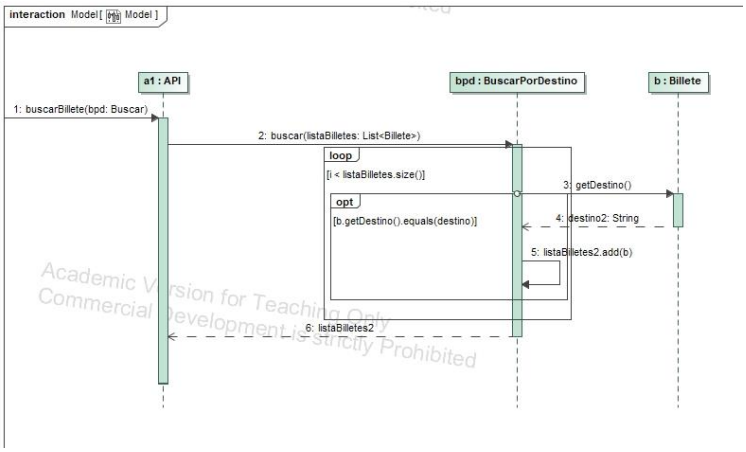
Patrón utilizado:

- Patrón Composición:
  Para este primer ejercicio hemos elegido el patrón composición, que se utiliza para componer objetos en estructuras de árbol que representan jerarquías todo-parte. Es el que más se adecúa a las necesidades del programa, puesto que necesitamos varios criterios concretos de búsqueda y, además, se puede pedir también una combinación de los mismos. Esto coincide con el patrón, pues consiste básicamente en un componente, unos componentes concretos y una composición, tratando los objetos básicos y los objetos compuestos de forma uniforme. Además, nos facilita la introducción de nuevos criterios de búsqueda en el futuro.

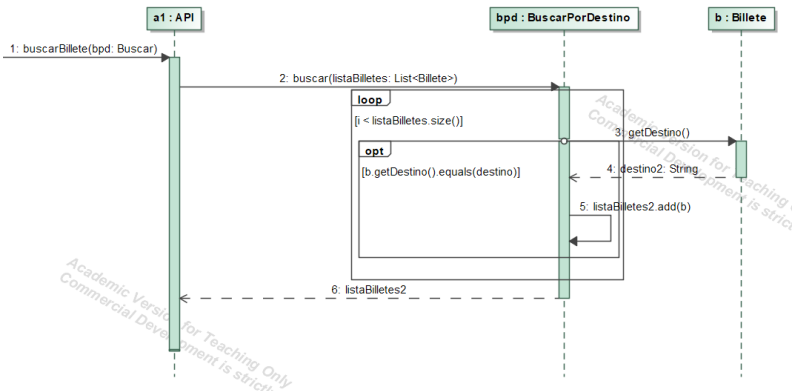
- Diagrama de clases



- Diagramas dinámicos



(Ejemplo de buscar por destino)



Ejercicio 2.

Principios utilizados:

- Responsabilidad única:
  Se fundamenta en que cada objeto tenga una única responsabilidad que esté enteramente encapsulada en la clase. En este caso, lo utilizamos en las clases DependenciaFuerte, DependenciaDebil y OrdenJerarquico, puesto que su única función es realizar las tareas siguiendo un algoritmo en concreto.

- Abierto-cerrado:
  Este principio se basa en que al modificar lo que hace una clase no se afecte a su código. Así, en un futuro será muy sencillo introducir cualquier variación. En nuestro caso, se consigue con el uso de la interfaz RealizacionTareas, puesto que, si en el futuro quisiéramos añadir una nueva forma de realizar las tareas, únicamente habría que introducir una nueva implementación de la interfaz.

- Inversión de la dependencia:
  Consiste en que los módulos de alto nivel no dependan de los módulos inferiores sino de abstracciones. Utilizamos este principio cuando declaramos una variable como RealizacionTareas y podemos utilizar cualquiera de las implementaciones (DependenciaFuerte, DependenciaDebil o OrdenJerarquico). Un ejemplo sería: RealizacionTareas realizar = new DependenciaFuerte().

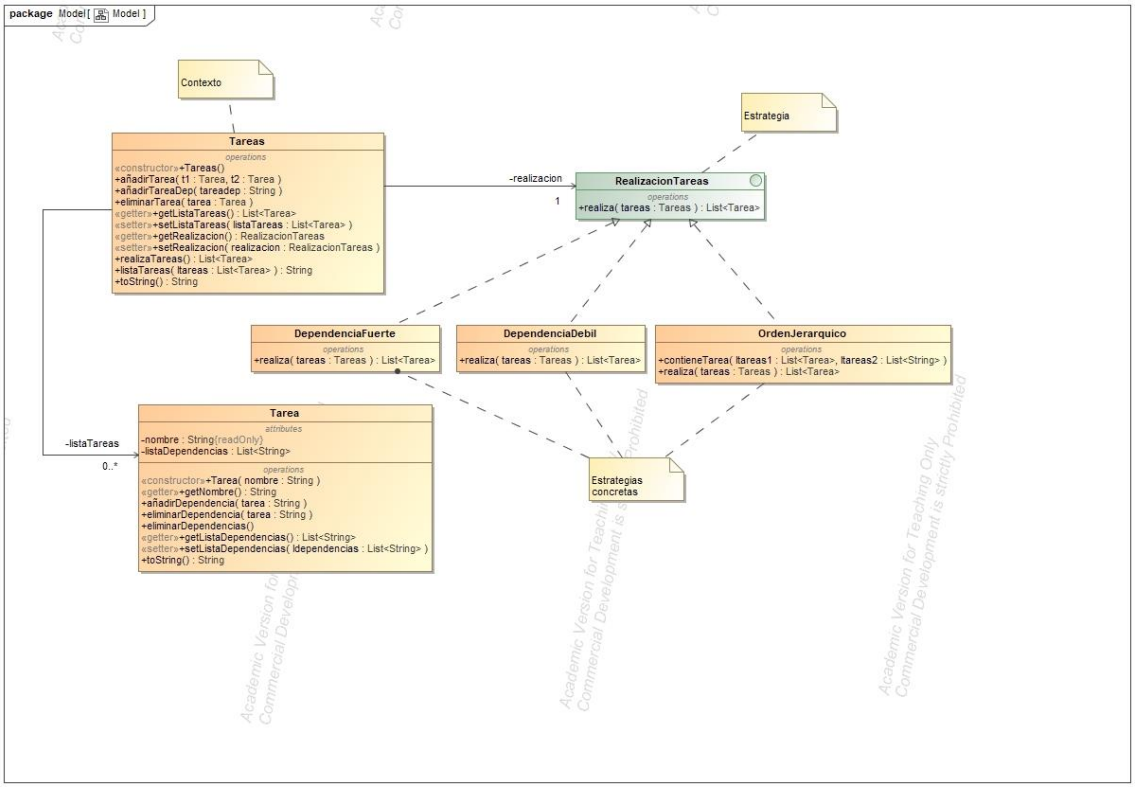
- Encapsula lo que varía:
  Se basa en identificar los aspectos de la aplicación que varían y separarlos de aquellos que permanecen estables. Lo utilizamos al separar las distintas formas de realizar las tareas según sus dependencias.

- Bajo acoplamiento:
  Se centra en buscar diseños débilmente acoplados entre objetos que interactúan. Lo utilizamos al crear objetos cohesivos, centrados en resolver una única responsabilidad (DependenciaFuerte, DependenciaDebil o OrdenJerarquico), con bajo acoplamiento.

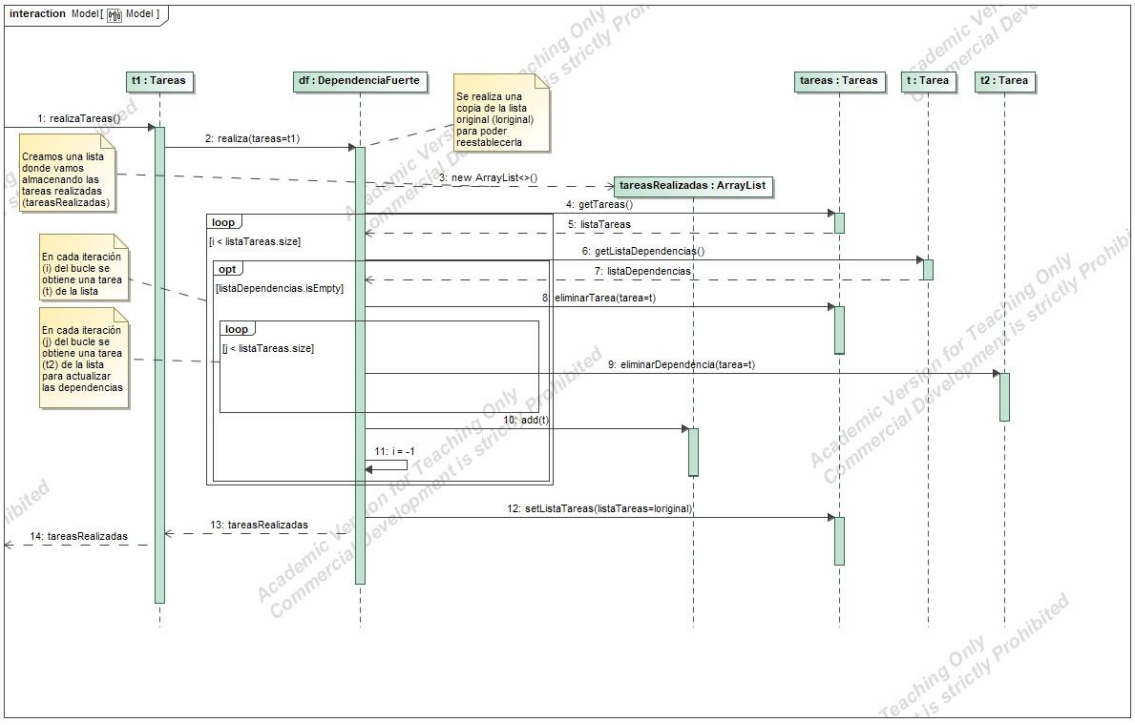
Patrón utilizado:

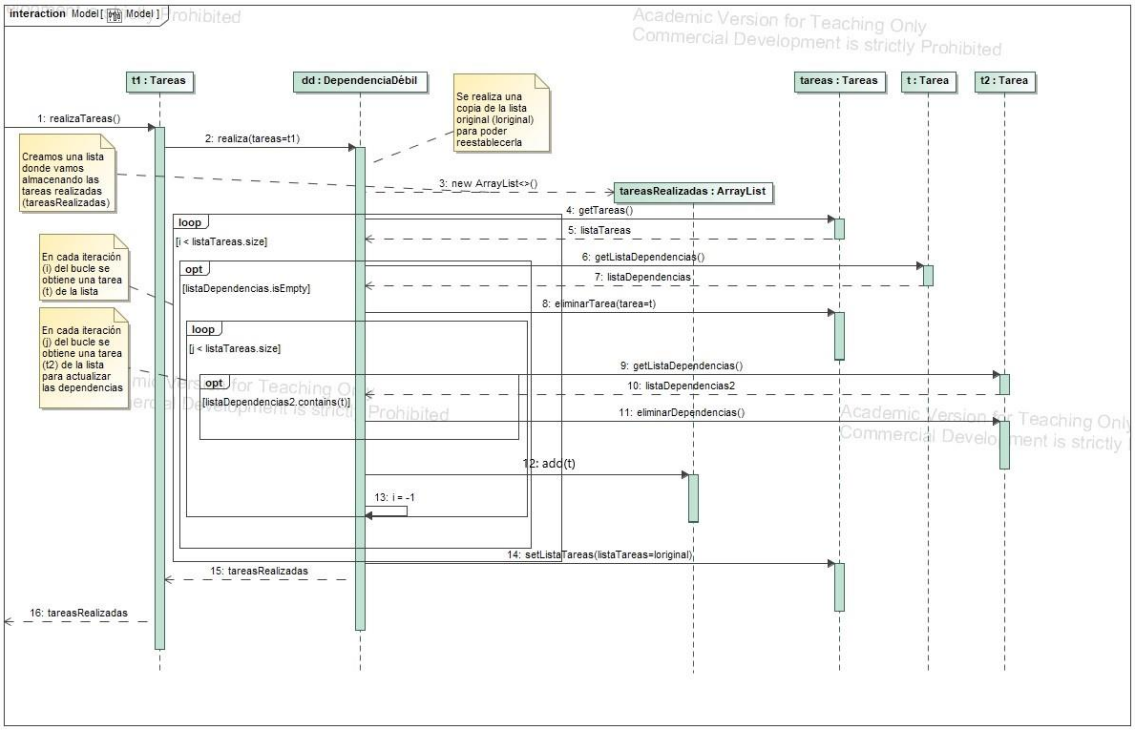
- Patrón Estrategia:
  Para este segundo ejercicio hemos elegido el patrón estrategia, que se utiliza para definir una familia de algoritmos, encapsularlos y hacerlos intercambiables. Este patrón resuelve satisfactoriamente las necesidades del problema propuesto, puesto que tenemos distintas formas de realizar las tareas (y pueden aparecer nuevas en el futuro) y este nos permite definir estas opciones de realización (DependenciaFuerte, DependenciaDebil y OrdenJerarquico) y encapsularlas. Además nos permite cambiar fácilmente de una forma de realizar las tareas a otra sin tener que utilizar múltiples sentencias condicionales para elegir la opción adecuada de llevarlas a cabo según las condiciones del entorno.

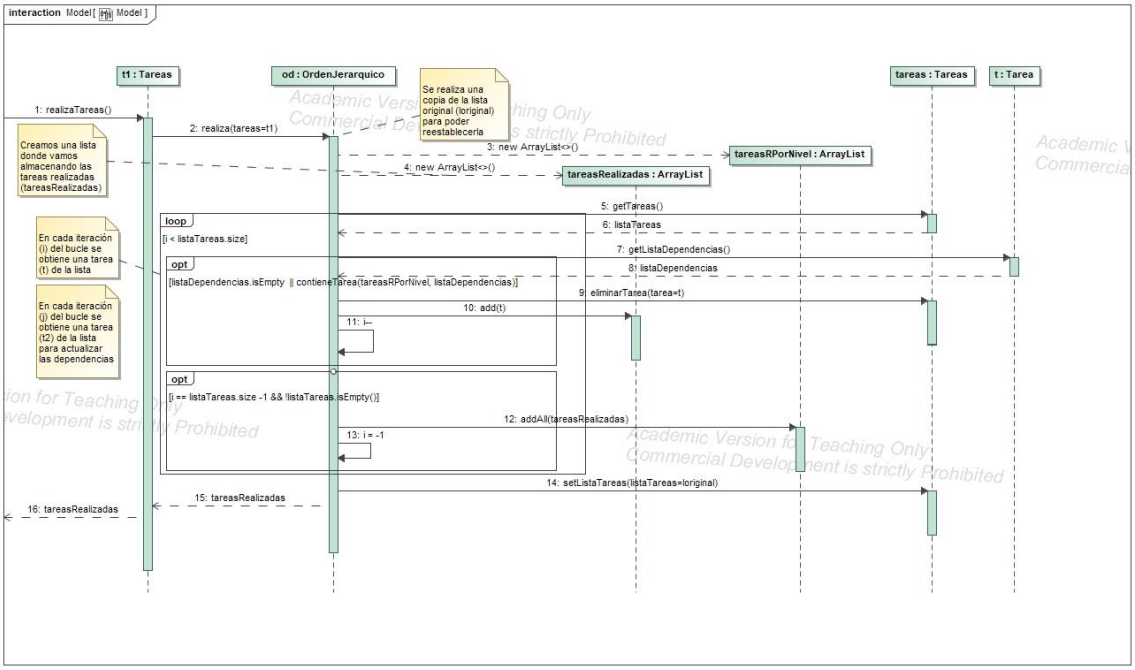
- Diagrama de clases



- Diagramas dinámicos




interaction Model [Model]
t1 : Tareas
dd : DependenciaDébil
Se realiza una copia de la lista original (loriginal) para poder reestablecerla
tareas : Tareas
t : Tarea
t2 : Tarea
1: realizaTareas()
2: realiza(tareas=t1)
Creamos una lista donde vamos almacenando las tareas realizadas (tareasRealizadas)
3: new ArrayList<>()
tareasRealizadas : ArrayList
4: getTareas()
5: listaTareas
loop
[i < listaTareas.size]
6: getListaDependencias()
En cada iteración (i) del bucle se obtiene una tarea (t) de la lista
opt
[listaDependencias.isEmpty]
7: listaDependencias
8: eliminarTarea(tarea=t)
En cada iteración (j) del bucle se obtiene una tarea (t2) de la lista para actualizar las dependencias
loop
[j < listaTareas.size]
9: getListaDependencias()
opt
[listaDependencias2.contains(t)]
10: listaDependencias2
11: eliminarDependencias()
12: add(t)
13: i = -1
14: setListaTareas(listaTareas=loriginal)
15: tareasRealizadas
16: tareasRealizadas


interaction Model [Model]
t1 : Tareas
od : OrdenJerarquico
Se realiza una copia de la lista original (loriginal) para poder reestablecerla
tareas : Tareas
t : Tarea
1: realizaTareas()
2: realiza(tareas=t1)
Creamos una lista donde vamos almacenando las tareas realizadas (tareasRealizadas)
3: new ArrayList<>()
tareasRPorNivel : ArrayList
4: new ArrayList<>()
tareasRealizadas : ArrayList
5: getTareas()
loop
[i < listaTareas.size]
6: listaTareas
7: getListaDependencias()
En cada iteración (i) del bucle se obtiene una tarea (t) de la lista
opt
[listaDependencias.isEmpty || contieneTarea(tareasRPorNivel, listaDependencias)]
8: listaDependencias
9: eliminarTarea(tarea=t)
10: add(t)
En cada iteración (j) del bucle se obtiene una tarea (t2) de la lista para actualizar las dependencias
11: i--
opt
[i == listaTareas.size -1 && listaTareas.isEmpty()]
12: addAll(tareasRealizadas)
13: i = -1
14: setListaTareas(listaTareas=loriginal)
15: tareasRealizadas
16: tareasRealizadas